Aerospace Blockset™

User's Guide

MATLAB® & SIMULINK®

R2024a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Aerospace Blockset™ User's Guide

Revision History

July 2002	Online only	New for Version 1.0 (Release 13)
July 2003	Online only	Revised for Version 1.5 (Release 13SP1)
June 2004	Online only	Revised for Version 1.6 (Release 14)
October 2004	Online only	Revised for Version 1.6.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.6.2 (Release 14SP2)
May 2005	Online only	Revised for Version 2.0 (Release 14SP2+)
September 2005	First printing	Revised for Version 2.0.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.1 (Release 2006a)
September 2006	Online only	Revised for Version 2.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.3 (Release 2007a)
September 2007	Second printing	Revised for Version 3.0 (Release 2007b)
March 2008	Online only	Revised for Version 3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.5 (Release 2010a)
September 2010	Online only	Revised for Version 3.6 (Release 2010b)
April 2011	Online only	Revised for Version 3.7 (Release 2011a)
September 2011	Online only	Revised for Version 3.8 (Release 2011b)
March 2012	Online only	Revised for Version 3.9 (Release 2012a)
September 2012	Online only	Revised for Version 3.10 (Release 2012b)
March 2013	Online only	Revised for Version 3.11 (Release 2013a)
September 2013	Online only	Revised for Version 3.12 (Release 2013b)
March 2014	Online only	Revised for Version 3.13 (Release 2014a)
October 2014	Online only	Revised for Version 3.14 (Release 2014b)
March 2015	Online only	Revised for Version 3.15 (Release 2015a)
September 2015	Online only	Revised for Version 3.16 (Release 2015b)
October 2015	Online only	Rereleased for Version 3.15.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.17 (Release 2016a)
September 2016	Online only	Revised for Version 3.18 (Release 2016b)
March 2017	Online only	Revised for Version 3.19 (Release 2017a)
September 2017	Online only	Revised for Version 3.20 (Release 2017b)
March 2018	Online only	Revised for Version 3.21 (Release 2018a)
September 2018	Online only	Revised for Version 4.0 (Release 2018b)
March 2019	Online only	Revised for Version 4.1 (Release 2019a)
September 2019	Online only	Revised for Version 4.2 (Release 2019b)
March 2020	Online only	Revised for Version 4.3 (Release 2020a)
September 2020	Online only	Revised for Version 4.4 (Release 2020b)
March 2021	Online only	Revised for Version 5.0 (Release 2021a)
September 2021	Online only	Revised for Version 5.1 (Release 2021b)
March 2022	Online only	Revised for Version 5.2 (Release 2022a)
September 2022	Online only	Revised for Version 5.3 (Release 2022b)
March 2023	Online only	Revised for Version 6.0 (Release 2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)

Contents

1 Getting Started

2 Aerospace Blockset Software

3 Case Studies

4 Supporting Data

A Aerospace Units Appendix

1

Getting Started

Aerospace Blockset Product Description

Model, simulate, and analyze aerospace vehicle dynamics

Aerospace Blockset provides Simulink® reference examples and blocks for modeling, simulating, and analyzing high-fidelity aircraft, rotocraft, and spacecraft platforms. The blockset includes vehicle dynamics, validated models of the flight environment, and blocks for pilot behavior, actuator dynamics, and propulsion. Built-in aerospace math operations and coordinate system and spatial transformations let you represent atmospheric vehicle and spacecraft motion and orientation. To examine simulation results, you can connect visualization blocks, including photorealistic views, to your model.

Aerospace Blockset provides standard model architectures for building reusable vehicle platform models. These models can support flight and mission analysis; conceptual studies; detailed mission design; guidance, navigation, and control (GNC) algorithm development; software integration testing; and hardware-in-the-loop (HIL) testing for applications in autonomous flight, radar, and communications.

Code Generation Support

Use the Aerospace Blockset software with the Simulink Coder software to automatically generate code for real-time execution in rapid prototyping and for hardware-in-the-loop systems.

Support for Aerospace Toolbox Quaternion Functions

The Aerospace Blockset product supports the following Aerospace Toolbox quaternion functions in the MATLAB Function block:

```
quatconj
quatinv
quatmod
quatmultiply
quatdivide
quatnorm
quatnormalize
```

For further information on using the MATLAB Function block, see:

- “Implement MATLAB Functions in Simulink with MATLAB Function Blocks”
- “Quaternion Estimate from Measured Rates” on page 9-55 example, which illustrates quaternions and models the equations

Explore the NASA HL-20 Model

In this section...

Introduction

This section introduces a NASA HL-20 lifting body airframe model that uses blocks from the Aerospace Blockset software to simulate the airframe of a NASA HL-20 lifting body, in conjunction with other Simulink blocks.

The model simulates the NASA HL-20 lifting body airframe approach and landing flight phases using an automatic-landing controller.

For more information on this model, see "NASA HL-20 Lifting Body Airframe" on page 3-14.

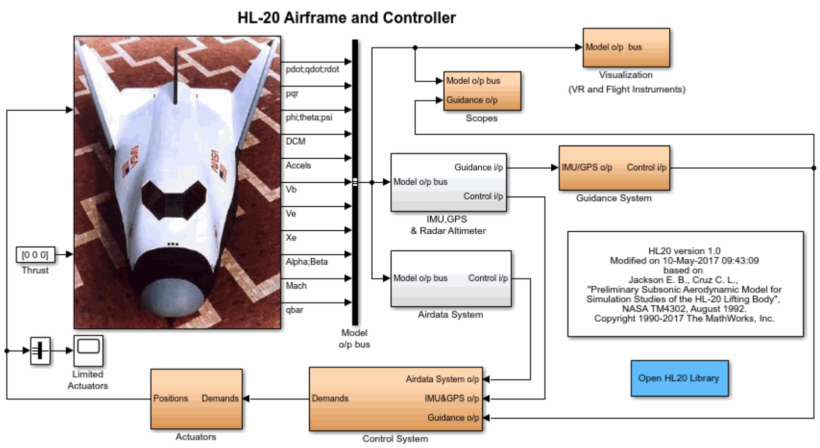
What This Example Illustrates

The NASA HL-20 lifting body airframe example illustrates the following features of the blockset:

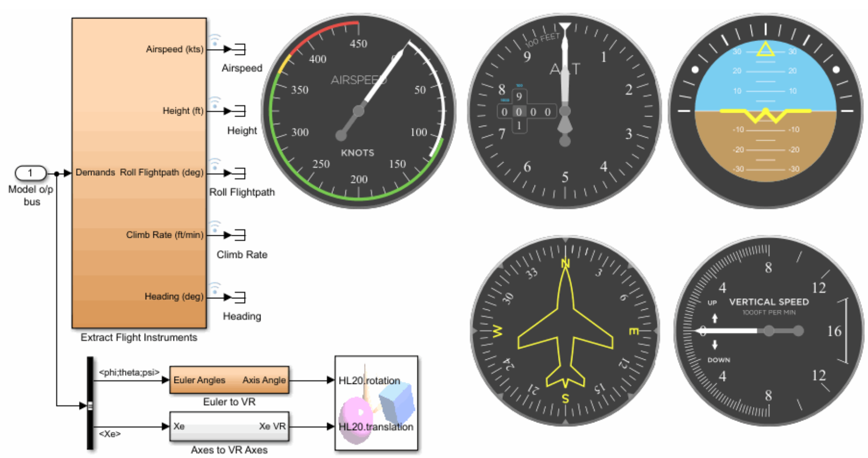
- Representing bodies and their degrees of freedom with the Equations of Motion library blocks
- Using the Aerospace Blockset blocks with other Simulink blocks
- Feeding Simulink signals to and from Aerospace Blockset blocks with Actuator and Sensor blocks
- Encapsulating groups of blocks into subsystems
- Visualizing an aircraft with Simulink 3D Animation™ and Aerospace Blockset Flight Instrument library blocks.

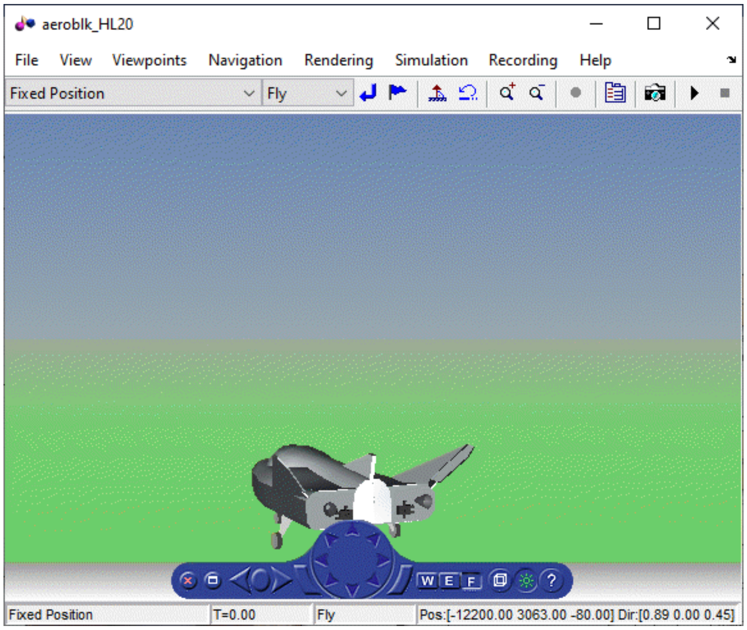
Open the Example

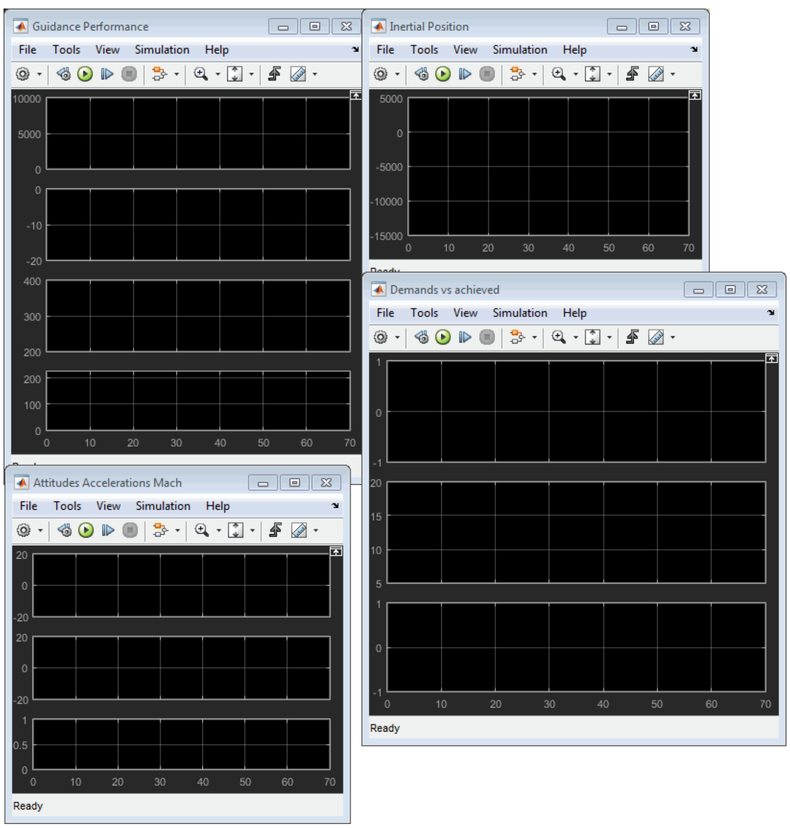
To open the NASA HL-20 airframe example, type the command `openExample('aeroblk_HL20_UE')`, at the MATLAB® command line. The model opens.



The visualization subsystem, multiple scopes, and a Simulink 3D Animation viewer for the airframe might also appear.







Key Subsystems

The model implements the airframe using the following subsystems:

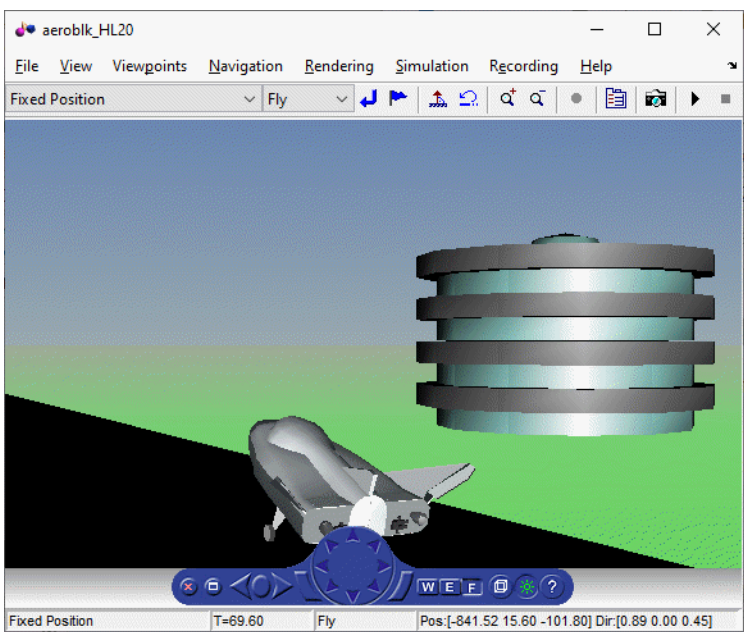
- The 6DOF (Euler Angles) subsystem implements the 6DOF (Euler Angles) block along with other Simulink blocks.
- The Environment Models subsystem implements the WGS84 Gravity Model and COESA Atmosphere Model blocks. It also contains a Wind Models subsystem that implements a number of wind blocks.
- The Alpha, Beta, Mach subsystem implements the Incidence, Sideslip, & Airspeed, Mach Number, and Dynamic Pressure blocks. These blocks calculate aerodynamic coefficient values and lookup functionality.
- The Forces and Moments subsystem implements the Aerodynamic Forces and Moments block. This subsystem calculates body forces and body moments.
- The Aerodynamic Coefficients subsystem implements several subsystems to calculate six aerodynamic coefficients.

NASA HL-20 Example

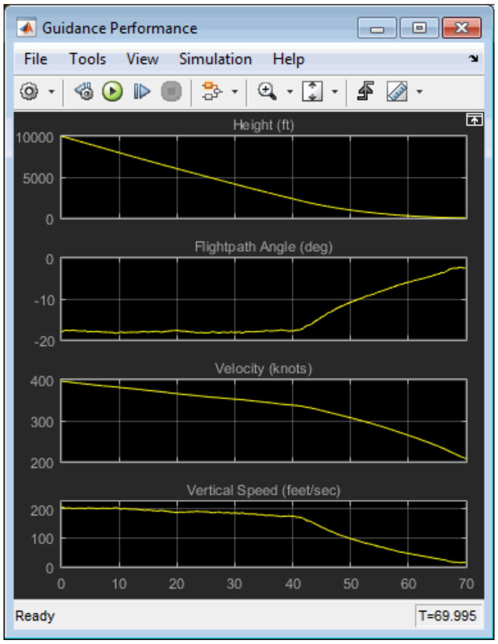
Running an example lets you observe the model simulation in real time. After you run the example, you can examine the resulting data in plots, graphs, and other visualization tools. To run this model, follow these steps:

1. If it is not already open, open the `aeroblk_HL20_UE` example.
2. In the Simulink Editor, from the **Simulation** tab, select **Run**.

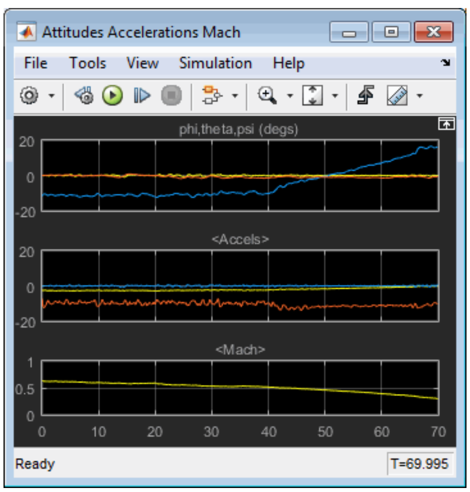
   The simulation proceeds until the aircraft lands:



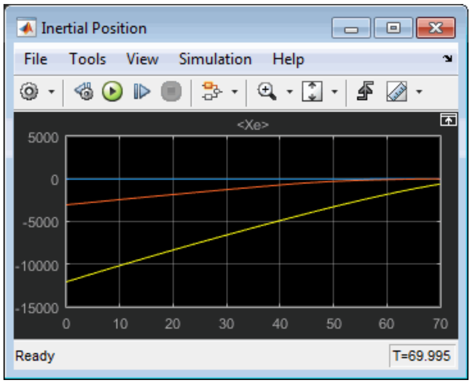
View of the landed airframe



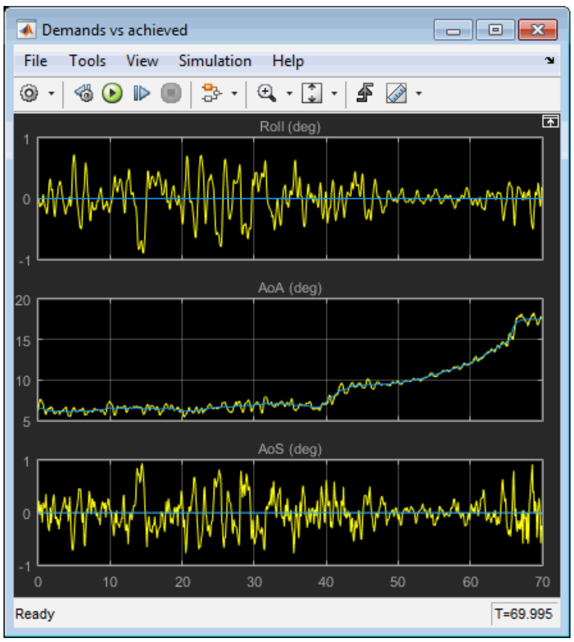
Plot that Measures Guidance Performance



Plot that Measures Altitude Accelerations Mach



Plot that Measures Inertial Position



Plot that Measures Demand Data Against Achieved Data

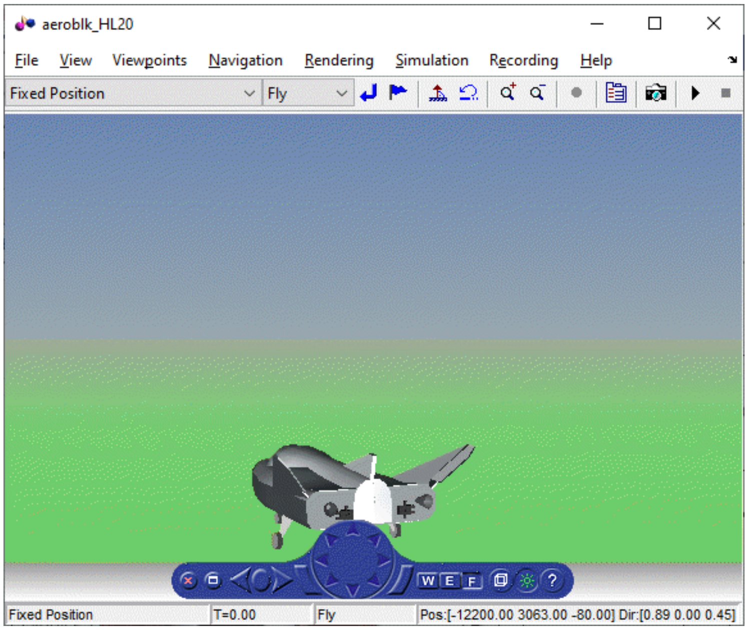
Modify the Model

You can adjust the airframe model settings and examine the effects on simulation performance. Here is one modification that you can try. It changes the camera point of view for the landing animation.

Change the Animation Point of View

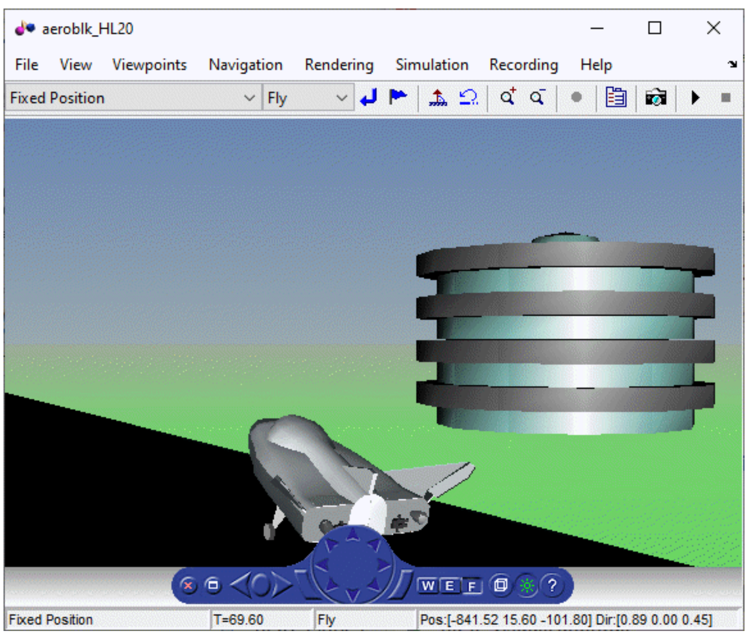
By default, the airframe animation viewpoint is `Rear position`, which means the view tracks with the airframe flight path from the rear. You can change the animation point of view by selecting another viewpoint from the Simulink 3D Animation viewer:

1. Open the `aeroblk_HL20_UE` model, and click the Simulink 3D Animation viewer.
2. From the list of existing viewpoints, change the viewpoint to `Fixed Position`.



The airframe view changes to a fixed position.

3 Start the model again. Notice the different airframe viewpoint when the airframe lands.



You can experiment with different viewpoints to watch the animation from different perspectives.

See Also

6DOF (Euler Angles) | Incidence, Sideslip, & Airspeed | Mach Number | Dynamic Pressure | Aerodynamic Forces and Moments

Related Examples

- “Flight Instrument Gauges” on page 2-58
- “Simulink 3D Animation Viewer” (Simulink 3D Animation)

Aerospace Blockset Software

Create Aerospace Models

In this section...

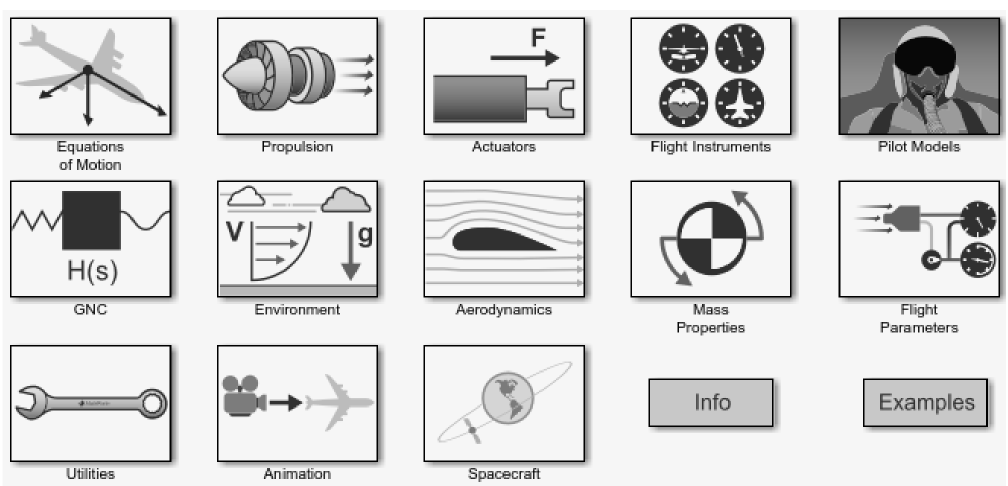
Basic Steps

Regardless of the model complexity, you use the same essential steps for creating an aerospace model as you would for creating any other Simulink model.

1. Open the Aerospace Blockset Library. You can access this library through the Simulink Library Browser or directly open the Aerospace Blockset window from the MATLAB command line:

   ```
   aerolib
   ```

   Double-click any library in the window to display its contents. This figure shows the Aerospace Blockset library window.

2. *Select and position the blocks.* You must first select the blocks that you need to build your model, and then position the blocks in the model window. For the majority of Simulink models, you select one or more blocks from each of these categories:

   a. Source blocks generate or import signals into the model, such as a sine wave, a clock, or limited-band white noise.

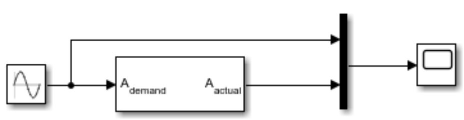
   b. Simulation blocks can consist of almost any type of block that performs an action in the simulation. A simulation block represents a part of the model functionality to be simulated, such as an actuator block, a mathematical operation, a block from the Aerospace Blockset library, and so on.

   c. Signal Routing blocks route signals from one point in a model to another. If you need to combine or redirect two or more signals in your model, you will probably use a Simulink Signal Routing block, such as Mux and Demux.

      As an alternative to the Mux block, consider the `Vector` option of the Vector Concatenate block **Mode** parameter. This block provides a more general way for you to route signals from one point in a model to another. The `Vector` mode takes as input a vector of signals of the same data type and creates a contiguous output signal. Depending on the input, this block outputs a row or column vector if any of the inputs are row or column vectors, respectively.
   d. Sink blocks display, write, or save model output. To see the results of the simulation, you must use a Sink block.
3. *Configure the blocks.* Most blocks feature configuration options that let you customize block functionality to specific simulation parameters. For example, the ISA Atmosphere Model block provides configuration options for setting the height of the troposphere, tropopause, and air density at sea level.
4. *Connect the blocks.* To create signal pathways between blocks, you connect the blocks to each other. You can do this manually by clicking and dragging, or you can connect blocks automatically.
5. *Encapsulate subsystems.* Systems made with Aerospace Blockset blocks can function as subsystems of larger, more complex models, like subsystems in any Simulink model.

Build a Simple Actuator System

The Simulink product is a software environment for modeling, simulating, and analyzing dynamic systems. Try building a simple model that drives an actuator with a sine wave and displays the actuator's position superimposed on the sine wave.



Note If you prefer to open the complete model instead of building it, see “Simple Actuator System Model” on page 2-4.

The “Create a Model” on page 2-4 section explains how to build a model on Windows® platforms. You can use this same procedure to build a model on Linux® platforms.

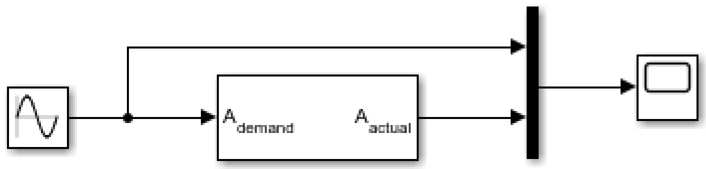
The section describes how to build the model. It does not describe how to set the configuration parameters for the model. See “Set Model Configuration Parameters for a Model”. That topic describes the Configuration Parameters dialog box for models. If you do not set any configuration parameters, simulating models might cause warnings like:

```
Warning: Using a default value of 0.2 for maximum step size.
The simulation step size will be equal to or less than this
value.  You can disable this diagnostic by setting
'Automatic solver parameter selection' diagnostic to 'none'
in the Diagnostics page of the configuration parameters
dialog
```

Simple Actuator System Model

This example shows how to build a simple actuator system model.

```
model='aeroblktutorial';
open_system(model)
```



Create a Model

To create a new blank model and open the Simulink library browser:

1. On the MATLAB **Home** tab, click Simulink. In the Simulink start page, click the Blank Model template, and then click Create Model.
2. To open the Library Browser, click the browser button.
3. Add a Sine Wave block to the model.

   a. Click **Sources** in the Library Browser to view the blocks in the Simulink Sources library.
   b. Drag the Sine Wave block from the Sources library into the new model window.
4. Add a Linear Second-Order Actuator block to the model.

   a. Click the ▷ symbol next to **Aerospace Blockset** in the Library Browser to expand the hierarchical list of the aerospace blocks.
   b. In the expanded list, click **Actuators** to view the blocks in the Actuator library.
   c. Drag the Linear Second-Order Actuator block into the model window.
5. Add a Mux block to the model.

   a. Click **Signal Routing** in the Library Browser to view the blocks in the Simulink Signals & Systems library.
   b. Drag the Mux block from the Signal Routing library into the model window.
6. Add a Scope block to the model.

   a. Click **Sinks** in the Library Browser to view the blocks in the Simulink Sinks library.
   b. Drag the Scope block from the Sinks library into the model window.
7. Resize the Mux block in the model.

   a. Click the Mux block to select the block.
   b. Hold down the mouse button and drag a corner of the Mux block to change the size of the block.

8 Connect the blocks.

   a Position the pointer near the output port of the Sine Wave block. Hold down the mouse button and drag the line that appears until it touches the input port of the Linear Second-Order Actuator block. Release the mouse button.

   b Using the same technique, connect the output of the Linear Second-Order Actuator block to the second input port of the Mux block.

   c Using the same technique, connect the output of the Mux block to the input port of the Scope block.

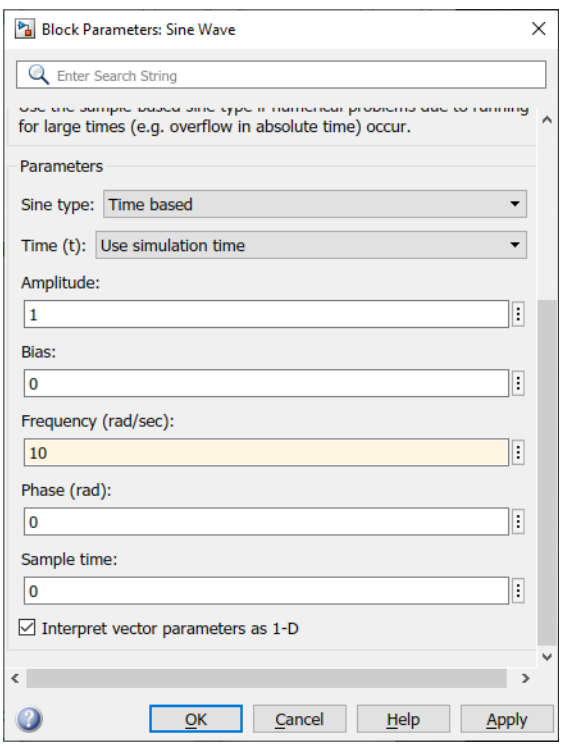
   d Position the pointer near the first input port of the Mux block. Hold down the mouse button and drag the line that appears over the line from the output port of the Sine Wave block until double crosshairs appear. Release the mouse button. The lines are connected when a knot is present at their intersection.

9 Set the block parameters.

   a Double-click the Sine Wave block. The dialog box that appears allows you to set the block's parameters.

   For this example, configure the block to generate a 10 rad/s sine wave by entering `10` for the **Frequency** parameter. The sinusoid has the default amplitude of `1` and phase of `0` specified by the **Amplitude** and **Phase offset** parameters.

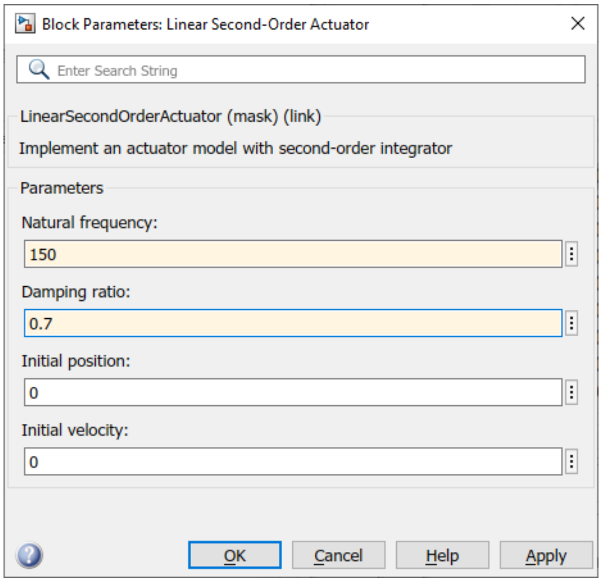
   b Click **OK**.



   c Double-click the Linear Second-Order Actuator block.

   In this example, the actuator has the default natural frequency of `150` rad/s, a damping ratio of `0.7`, and an initial position of `0` radians specified by the **Natural frequency**, **Damping ratio**, and **Initial position** parameters.

d Click **OK.**



Run the Simulation

You can now run the model that you built to see how the system behaves in time:

1. Double-click the Scope block if the Scope window is not already open on your screen. The Scope window appears.
2. Select **Run** from the **Simulation** menu in the model window. The signal containing the 10 rad/s sinusoid and the signal containing the actuator position are plotted on the scope.
3. Adjust the Scope block's display. While the simulation is running, right-click the *y*-axis of the scope and select **Autoscale**. The vertical range of the scope is adjusted to better fit the signal.
4. Vary the Sine Wave block parameters.

   a While the simulation is running, double-click the Sine Wave block to open its parameter dialog box.

   b You can then change the frequency of the sinusoid. Try entering `1` or `20` in the **Frequency** field. Close the Sine Wave dialog box to enter your change. You can then observe the changes on the scope.
5. Select **Stop** from the **Simulation** menu to stop the simulation.

Many parameters *cannot* be changed while a simulation is running. This is usually the case for parameters that directly or indirectly alter a signal's dimensions or sample rate. However, there are some parameters, like the Sine Wave **Frequency** parameter, that you can *tune* without stopping the simulation.

Run a Simulation from a Script

You can also modify and run a Simulink simulation from a script. By doing this, you can automate the variation of model parameters to explore a large number of simulation conditions rapidly and efficiently. For information on how to do this, see “Run Simulations Programmatically”.

Access Aerospace Examples

To access an Aerospace Blockset example:

1. Open the MATLAB Command Window.
2. Click the question mark.
3. Navigate to Aerospace Blockset and click the **Examples** tab.

See Also

Linear Second-Order Actuator

Related Examples

- “Run Simulations Programmatically”
- “Ideal Airspeed Correction” on page 3-2
- “1903 Wright Flyer” on page 3-7
- “NASA HL-20 Lifting Body Airframe” on page 3-14

About Aerospace Coordinate Systems

In this section...

Fundamental Coordinate System Concepts

Coordinate systems allow you to keep track of an aircraft or spacecraft position and orientation in space. The Aerospace Blockset coordinate systems are based on these underlying concepts from geodesy, astronomy, and physics.

Definitions

The blockset uses right-handed (RH) Cartesian coordinate systems. The right-hand rule establishes the *x*-*y*-*z* sequence of coordinate axes.

An inertial frame is a nonaccelerating motion reference frame. In an inertial frame, Newton's second law holds: force = mass x acceleration. Loosely speaking, acceleration is defined with respect to the distant cosmos, and an inertial frame is often said to be nonaccelerated with respect to the fixed stars. Because the Earth and stars move so slowly with respect to one another, this assumption is a very accurate approximation.

Strictly defined, an inertial frame is a member of the set of all frames not accelerating relative to one another. A noninertial frame is any frame accelerating relative to an inertial frame. Its acceleration, in general, includes both translational and rotational components, resulting in pseudoforces (pseudogravity, as well as Coriolis and centrifugal forces).

The blockset models the Earth shape (the geoid) as an oblate spheroid, a special type of ellipsoid with two longer axes equal (defining the equatorial plane) and a third, slightly shorter (geopolar) axis of symmetry. The equator is the intersection of the equatorial plane and the Earth surface. The geographic poles are the intersection of the Earth surface and the geopolar axis. In general, the Earth geopolar and rotation axes are not identical.

Latitudes parallel the equator. Longitudes parallel the geopolar axis. The zero longitude or prime meridian passes through Greenwich, England.

Approximations

The blockset makes three standard approximations in defining coordinate systems relative to the Earth.

- The Earth surface or geoid is an oblate spheroid, defined by its longer equatorial and shorter geopolar axes. In reality, the Earth is slightly deformed with respect to the standard geoid.
- The Earth rotation axis and equatorial plane are perpendicular, so that the rotation and geopolar axes are identical. In reality, these axes are slightly misaligned, and the equatorial plane wobbles as the Earth rotates. This effect is negligible in most applications.

- The only noninertial effect in Earth-fixed coordinates is due to the Earth rotation about its axis. This is a rotating, geocentric system. The blockset ignores the Earth acceleration around the Sun, the Sun acceleration in the Galaxy, and the Galaxy acceleration through the cosmos. In most applications, only the Earth rotation matters.

  This approximation must be changed for spacecraft sent into deep space, such as outside the Earth-Moon system, and a heliocentric system is preferred.

Passive Transformations

All quaternions in Aerospace Blockset are passive transformations. In a passive transformation, the vector is unchanged and the coordinate system in which it is defined is rotated. For more information on transformations, see Active and passive transformations.

Motion with Respect to Other Planets

The blockset uses the standard WGS-84 geoid to model the Earth. You can change the equatorial axis length, the flattening, and the rotation rate.

You can represent the motion of spacecraft with respect to any celestial body that is well approximated by an oblate spheroid by changing the spheroid size, flattening, and rotation rate. If the celestial body is rotating westward (retrogradely), make the rotation rate negative.

Coordinate Systems for Modeling

Modeling aircraft and spacecraft is simplest if you use a coordinate system fixed in the body itself. In the case of aircraft, the forward direction is modified by the presence of wind, and the craft motion through the air is not the same as its motion relative to the ground.

See “Equations of Motion” for further details on how the blockset implements body and wind coordinates.

Body Coordinates

The noninertial body coordinate system is fixed in both origin and orientation to the moving craft. The craft is assumed to be rigid.

The orientation of the body coordinate axes is fixed in the shape of body.

- The x-axis points through the nose of the craft.
- The y-axis points to the right of the x-axis (facing in the pilot's direction of view), perpendicular to the x-axis.
- The z-axis points down through the bottom the craft, perpendicular to the xy plane and satisfying the RH rule.

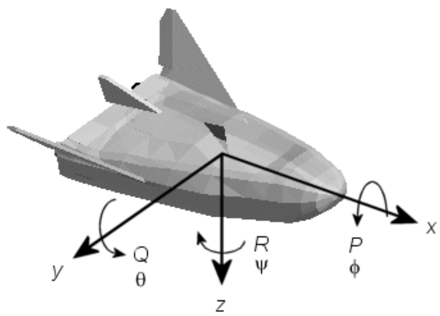
Translational Degrees of Freedom

Translations are defined by moving along these axes by distances x, y, and z from the origin.

Rotational Degrees of Freedom

Rotations are defined by the Euler angles P, Q, R or Φ, Θ, Ψ. They are:

P or Φ	Roll about the *x*-axis
Q or Θ	Pitch about the *y*-axis
R or Ψ	Yaw about the *z*-axis



Unless otherwise specified, by default the software uses ZYX rotation order for Euler angles.

Wind Coordinates

The noninertial wind coordinate system has its origin fixed in the rigid aircraft. The coordinate system orientation is defined relative to the craft velocity ***V***.

The orientation of the wind coordinate axes is fixed by the velocity ***V***.

- The *x*-axis points in the direction of ***V***.
- The *y*-axis points to the right of the *x*-axis (facing in the direction of ***V***), perpendicular to the *x*-axis.
- The *z*-axis points perpendicular to the *xy* plane in whatever way needed to satisfy the RH rule with respect to the *x*- and *y*-axes.

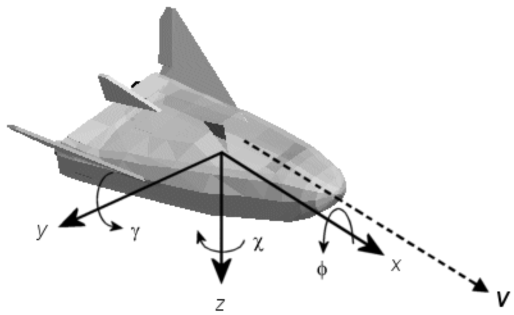
Translational Degrees of Freedom

Translations are defined by moving along these axes by distances *x*, *y*, and *z* from the origin.

Rotational Degrees of Freedom

Rotations are defined by the Euler angles Φ, γ, χ:

Φ	Bank angle about the *x*-axis
γ	Flight path about the *y*-axis
χ	Heading angle about the *z*-axis



Unless otherwise specified, by default the software uses ZYX rotation order for Euler angles.

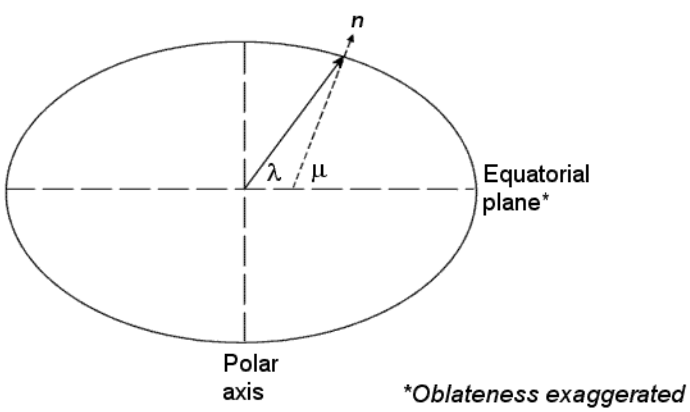
Coordinate Systems for Navigation

Modeling aerospace trajectories requires positioning and orienting the aircraft or spacecraft with respect to the rotating Earth. Navigation coordinates are defined with respect to the center and surface of the Earth.

Geocentric and Geodetic Latitudes

The geocentric latitude λ on the Earth surface is defined by the angle subtended by the radius vector from the Earth center to the surface point with the equatorial plane.

The geodetic latitude μ on the Earth surface is defined by the angle subtended by the surface normal vector n and the equatorial plane.

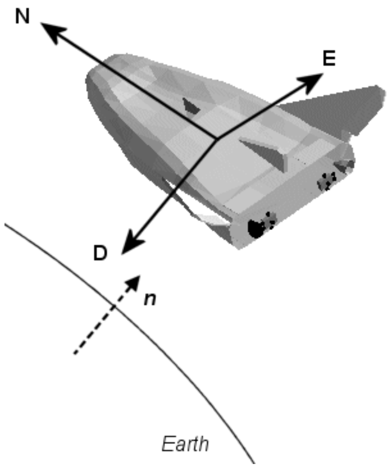


NED Coordinates

The north-east-down (NED) system is a noninertial system with its origin fixed at the aircraft or spacecraft center of gravity. Its axes are oriented along the geodetic directions defined by the Earth surface.

- The *x*-axis points north parallel to the geoid surface, in the polar direction.
- The *y*-axis points east parallel to the geoid surface, along a latitude curve.
- The *z*-axis points downward, toward the Earth surface, antiparallel to the surface outward normal ***n***.

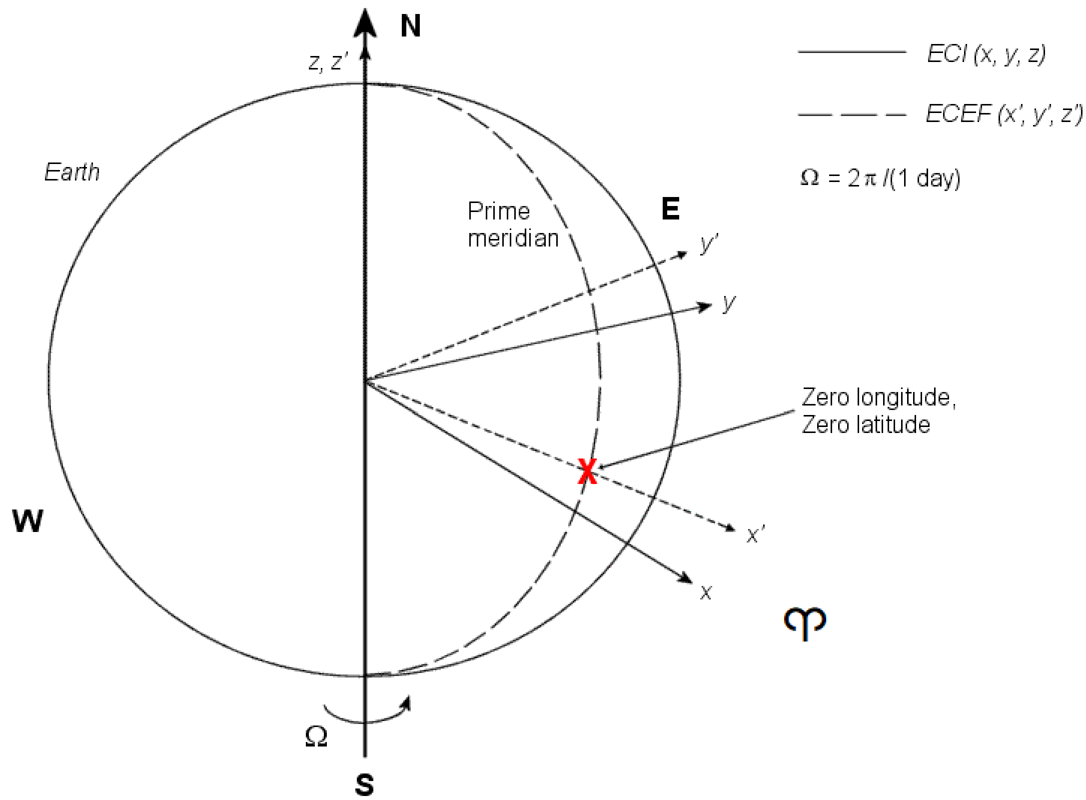
Flying at a constant altitude means flying at a constant z above the Earth surface.



ECI Coordinates

The Earth-centered inertial (ECI) system is non-rotating. For most applications, assume this frame to be inertial, although the equinox and equatorial plane move very slightly over time. The ECI system is considered to be truly inertial for high-precision orbit calculations when the equator and equinox are defined at a particular epoch (e.g. J2000). Aerospace functions and blocks that use a particular realization of the ECI coordinate system provide that information in their documentation. The ECI system origin is fixed at the center of the Earth (see figure).

- The x-axis points towards the vernal equinox (First Point of Aries ♈).
- The y-axis points 90 degrees to the east of the x-axis in the equatorial plane.
- The z-axis points northward along the Earth rotation axis.



Earth-Centered Coordinates

ECEF Coordinates

The Earth-center, Earth-fixed (ECEF) system is noninertial and rotates with the Earth. Its origin is fixed at the center of the Earth (see preceding figure).

- The *x′*-axis points towards the intersection of Earth equatorial plane and the Greenwich Meridian.
- The *y′*-axis points 90 degrees to the east of the *x′*-axis in the equatorial plane.
- The *z′*-axis points northward along the Earth rotation axis.

Coordinate Systems for Display

Several display tools are available for use with the Aerospace Blockset product. Each has a specific coordinate system for rendering motion.

MATLAB Graphics Coordinates

See the “Axes Appearance” for more information about the MATLAB Graphics coordinate axes.

MATLAB Graphics uses this default coordinate axis orientation:

- The *x*-axis points out of the screen.
- The *y*-axis points to the right.
- The *z*-axis points up.

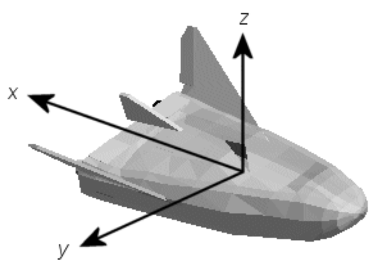
FlightGear Coordinates

FlightGear is an open-source, third-party flight simulator with an interface supported by the blockset.

- "Work with the Flight Simulator Interface" on page 2-24 discusses the blockset interface to FlightGear.
- See the FlightGear documentation at `www.flightgear.org` for complete information about this flight simulator.

The FlightGear coordinates form a special body-fixed system, rotated from the standard body coordinate system about the *y*-axis by -180 degrees:

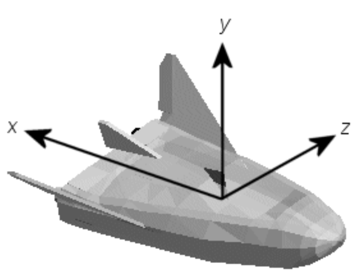
- The *x*-axis is positive toward the back of the vehicle.
- The *y*-axis is positive toward the right of the vehicle.
- The *z*-axis is positive upward, e.g., wheels typically have the lowest *z* values.



AC3D Coordinates

AC3D is a low-cost, widely used, geometry editor available from `https://www.inivis.com`. Its body-fixed coordinates are formed by inverting the three standard body coordinate axes:

- The *x*-axis is positive toward the back of the vehicle.
- The *y*-axis is positive upward, e.g., wheels typically have the lowest *y* values.
- The *z*-axis is positive to the left of the vehicle.



References

[1] *Recommended Practice for Atmospheric and Space Flight Vehicle Coordinate Systems*, R-004-1992, ANSI/AIAA, February 1992.

[2] Rogers, R. M., *Applied Mathematics in Integrated Navigation Systems,* AIAA, Reston, Virginia, 2000.

[3] Sobel, D., *Longitude*, Walker & Company, New York, 1995.

[4] Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation,* 2nd ed., *Aircraft Control and Simulation,* Wiley-Interscience, New York, 2003.

[5] Thomson, W. T., *Introduction to Space Dynamics,* John Wiley & Sons, New York, 1961/Dover Publications, Mineola, New York, 1986.

See Also

External Websites

- Office of Geomatics

Visualization Tools

You can visualize stages of your Aerospace Blockset application using flight simulator, flight instrument, aircraft scenario, and MATLAB graphics-based tools.

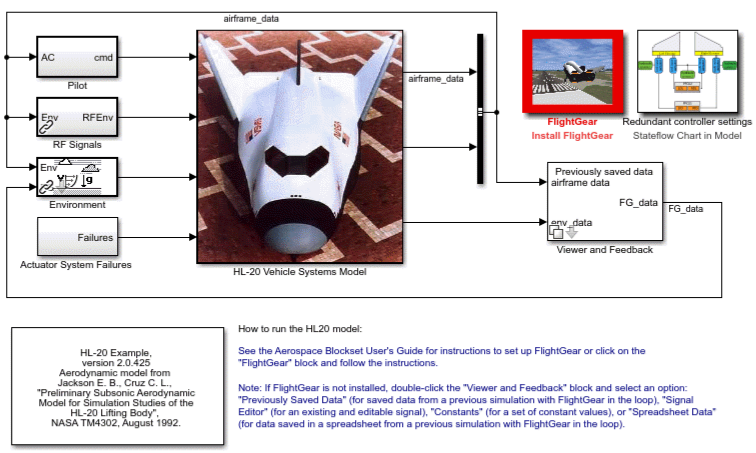
Flight Simulation Interface Blocks

Use flight simulator interface blocks to visualize flight paths using FlightGear and input FlightGear data to models.

To use FlightGear to visualize flight paths, first install FlightGear software. For more information, see "Flight Simulator Interface" on page 2-20. The Flight Simulator Interfaces sublibrary contains:

- FlightGear Preconfigured 6DoF Animation — Connect model to FlightGear flight simulator.
- Generate Run Script — Generate FlightGear run script on current platform.
- Pack net_fdm Packet for FlightGear — Generate net_fdm packet for FlightGear.
- Receive net_ctrl Packet from FlightGear — Receive net_ctrl packet from FlightGear.
- Send net_fdm Packet to FlightGear — Transmit net_fdm packet to destination IP address and port for FlightGear session.
- Unpack net_ctrl Packet from FlightGear — Unpack net_ctrl variable packet received from FlightGear.

Here is an example of the model in "HL-20 Project with Optional FlightGear Interface" on page 9-53.



Flight Instrument Blocks

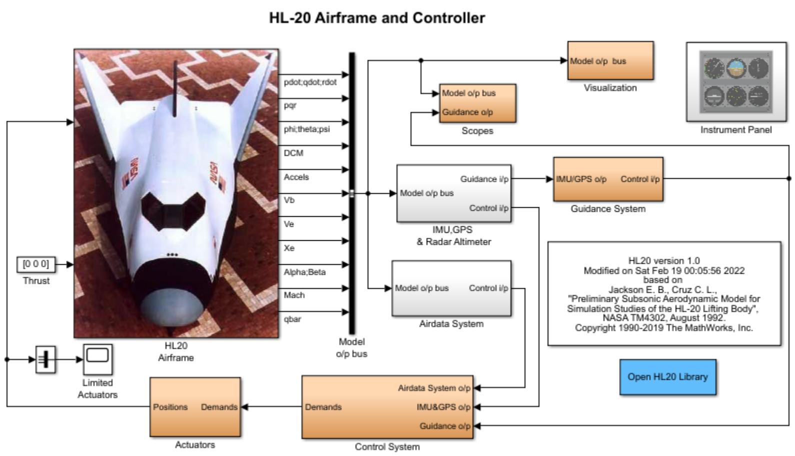
Use blocks representing standard cockpit instruments to display flight status information from the model. Each block graphically represents a cockpit instrument.

- Airspeed Indicator — Display measurements for aircraft airspeed
- Altimeter — Display measurements for aircraft altitude
- Artificial Horizon — Represent aircraft attitude relative to horizon
- Climb Rate Indicator — Display measurements for aircraft climb rate

- Exhaust Gas Temperature (EGT) Indicator — Display measurements for engine exhaust gas temperature (EGT)
- Heading Indicator — Display measurements for aircraft heading
- Revolutions Per Minute (RPM) Indicator — Display measurements for engine revolutions per minute (RPM) in percentage of RPM
- Turn Coordinator — Display measurements on turn coordinator and inclinometer

After you add a flight instruments block to a model, a table in the block dialog is automatically filled with observable signals from the model. To display one of the observable signals on an instrument gauge, select the signal.

Here is an example of the model with the flight instruments panel in “HL-20 with Flight Instrumentation Blocks” on page 9-36.



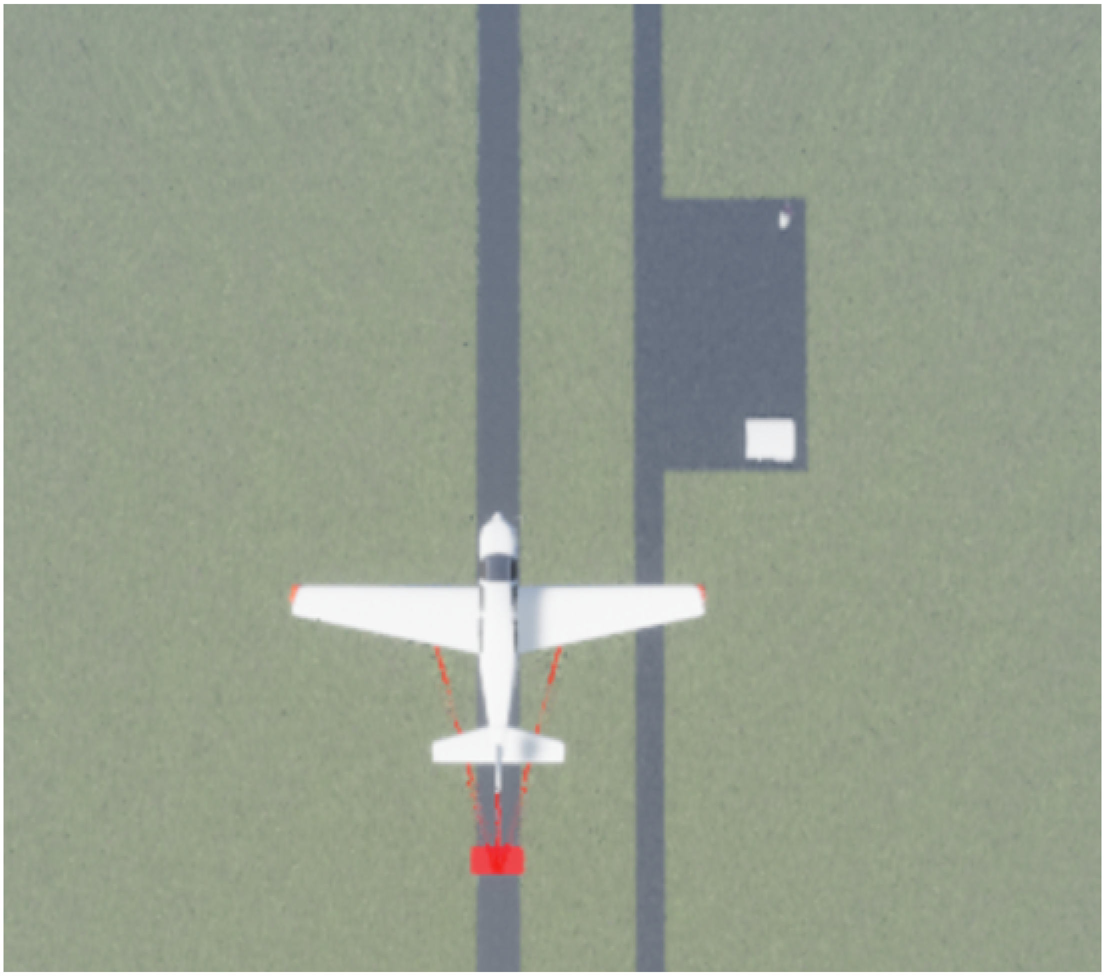
Simulation 3D Blocks

Use the Simulation 3D blocks to view 3D aircraft dynamics and airport operations in the Unreal® environment. Use supporting functions to customize your projects.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

- Simulation 3D Actor Transform Get — Get actor translation, rotation, scale.
- Simulation 3D Actor Transform Set — Set actor translation, rotation, scale.
- Simulation 3D Aircraft — Implement aircraft in 3D environment.
- Simulation 3D Camera Get — Camera image.
- Simulation 3D Scene Configuration — Scene configuration for 3D simulation environment.
- Simulation 3D Message Get — Retrieve data from Unreal Engine visualization environment.
- Simulation 3D Message Set — Send data to Unreal Engine visualization environment.

Here is the animation from a simulation in “Using Unreal Engine Visualization for Airplane Flight” on page 9-141.



To customize your projects, Aerospace Blockset also contains functions that let you work with blue print actors and send and receive messages between Unreal Engine and Simulink.

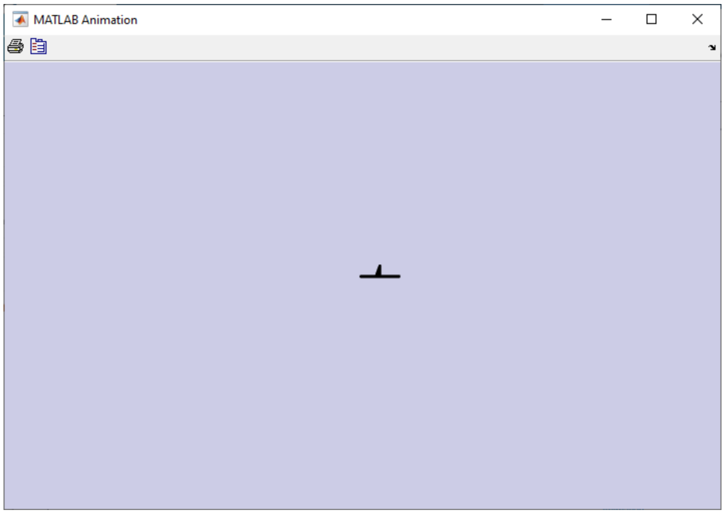
While the blocks and functions are available with Aerospace Blockset, to customize scenes in your installation of the Unreal Engine and simulate within these scenes in Simulink, you must install and configure the Aerospace Blockset Interface for Unreal Engine Projects support package. This support package contains customizable scenes and the Griffiss airport.

MATLAB Graphics-Based Animation

Use MATLAB graphics-based animation blocks to visualize flight paths using MATLAB graphics software.

- 3DoF Animation — Create 3-D MATLAB Graphics animation of three-degrees-of-freedom object.
- 6DoF Animation — Create 3-D MATLAB Graphics animation of six-degrees-of-freedom object.
- MATLAB Animation — Create six-degrees-of-freedom multibody custom geometry block.

Here is the animation from a simulation in “Multiple Aircraft with Collaborative Control” on page 9-33 using the MATLAB Animation block.



Utility Visualization Blocks

Use the utility visualization blocks to provide joystick interface inputs to your model. This library also contains the Simulation Pace block, to set the simulation rate for animation viewing.

- Pilot Joystick — Provide joystick interface on Windows platform. This block is the same as the Pilot Joystick All block with the **Output configuration** parameter set to `FourAxis`.
- Pilot Joystick All — Provide joystick interface in All Outputs configuration on Windows platform. This block is the same as the Pilot Joystick block with the **Output configuration** parameter set to `AllOutputs`.
- Simulation Pace — Set simulation rate for animation viewing.

See Also

Related Examples

- “Flight Simulator Interface” on page 2-20
- “Flight Instrument Gauges” on page 2-58
- “How 3D Simulation for Aerospace Blockset Works” on page 2-40

Flight Simulator Interface

About the FlightGear Interface

The Aerospace Blockset product supports an interface to the third-party FlightGear flight simulator, open-source software available through a GNU General Public License (GPL). The FlightGear flight simulator interface included with the blockset is a unidirectional transmission link from the Simulink interface to FlightGear using the FlightGear published `net_fdm` binary data exchange protocol. Data is transmitted via UDP network packets to a running instance of FlightGear. The blockset supports multiple standard binary distributions of FlightGear. See "Run FlightGear with Simulink Models" on page 2-28 for interface details.

FlightGear is a separate software entity not created, owned, or maintained by MathWorks®.

- To report bugs in or request enhancements to the Aerospace Blockset FlightGear interface, contact MathWorks technical support at https://www.mathworks.com/support/contact_us.html.
- To report bugs or request enhancements to FlightGear itself, visit `FlightGear website`.

Supported FlightGear Versions

The Aerospace Toolbox product supports FlightGear versions starting from v2.6.

If you are using a FlightGear version older than 2.6, update your FlightGear installation to a supported version. When you open the model, the software returns a warning or error. Obtain updated FlightGear software from `https://www.flightgear.org` in the download area.

Obtain FlightGear

You can obtain FlightGear from the FlightGear website in the download area or by ordering CDs from FlightGear. The download area contains extensive documentation for installation and configuration. Because FlightGear is an open-source project, source downloads are also available for customizing and porting to custom environments.

Configure Your Computer for FlightGear

To use FlightGear, you must have a high-performance graphics card with stable drivers For more information, see the FlightGear CD distribution or the hardware requirements and documentation areas of the FlightGear website.

FlightGear performance and stability can be sensitive to computer video cards, driver versions, and driver settings. You need OpenGL® support with hardware acceleration activated. Without proper setup, performance can drop from about a 30 frames-per-second (fps) update rate to less than 1 fps. If your system allows you to update OpenGL settings, modify them to improve performance.

Graphics Recommendations for Windows

For Windows systems, use these graphics recommendations:

- A graphics card with acceptable OpenGL performance (as outlined at the FlightGear website).
- The latest tested and stable driver release for your video card. Test the driver thoroughly on a few computers before deploying to others.

For more information, see `FlightGear Hardware Recommendations`.

Setup on Linux, Macintosh, and Other Platforms

FlightGear distributions are available for Linux, Macintosh, and other platforms from the FlightGear website, `https://www.flightgear.org`. Installation on these platforms, like Windows, requires careful configuration of graphics cards and drivers. Consult the documentation and hardware requirements sections at the FlightGear website.

FlightGear and Video Cards in Windows Systems

Your computer built-in video card, such as NVIDIA® cards, can conflict with FlightGear shaders. Consider this workaround:

- Disable the FlightGear shaders by selecting the Generate Run Script block **Disable FlightGear shader options** check box.

Install and Start FlightGear

The extensive FlightGear documentation guides you through the installation in detail. Consult these notes:

- Documentation section of the FlightGear website for installation instructions: `https://www.flightgear.org`.
- Hardware recommendations section of the FlightGear website.
- MATLAB system requirements.

Keep these points in mind:

- Configure your computer graphics card before you install FlightGear. See the preceding section, “Configure Your Computer for FlightGear” on page 2-20.
- Shut down all running applications (including the MATLAB interface) before installing FlightGear.
- Install FlightGear in a folder path name composed of ASCII characters.
- MathWorks tests indicate that the operational stability of FlightGear is especially sensitive during startup. It is best not to move, resize, mouse over, overlap, or cover up the FlightGear window until the initial simulation scene appears after the startup splash screen fades out.

Aerospace Blockset supports FlightGear on several platforms. This table lists the properties to consider before you start to use FlightGear.

FlightGear Property	Folder Description	Platforms	Typical Location
`FlightGearBase-Directory`	FlightGear installation folder.	Windows 64-bit	`C:\Program Files\FlightGear` (default)
		Linux	Folder into which you installed FlightGear

FlightGear Property	Folder Description	Platforms	Typical Location
		Mac	/Applications (folder to which you dragged the FlightGear icon)
GeometryModelName	Model geometry folder	Windows 64-bit	C:\Program Files\FlightGear\-data\Aircraft\HL20 (default)
		Linux	$FlightGearBaseDirectory/data/Aircraft/HL20
		Mac	$FlightGearBaseDirectory/-FlightGear.app/Contents/-Resources/data/Aircraft/HL20

Install Additional FlightGear Scenery

When you install the FlightGear software, the installation provides a basic level of scenery files. The FlightGear documentation guides you through installing scenery as part the general FlightGear installation.

If you need to install more FlightGear scenery files, see the instructions at `https://www.flightgear.org`. The instructions describe how to install the additional scenery in a default location. MathWorks recommends that you follow those instructions.

If you install additional scenery in a nonstandard location, you may need to update the `FG_SCENERY` environment variable in the script output from the Generate Run Script block to include the new path. For a description of the `FG_SCENERY` variable, see the documentation at https://www.flightgear.org.

If you do not download scenery in advance, you can direct FlightGear to download it automatically during simulation by selecting the Generate Run Script block **Install FlightGear scenery during simulation (requires Internet connection)** check box.

For Windows systems, you may encounter an error message while launching FlightGear with the `InstallScenery` option enabled:

```
Error creating directory: No such file or directory
```

This error likely indicates that your default FlightGear download folder is not writeable, the path cannot be resolved, or the path contains UNC path names. To work around the issue, edit the `runfg.bat` file to specify a new folder path to store the scenery data:

1. Edit `runfg.bat`.
2. To the list of command options, append `--download-dir=` and specify a folder to which to download the scenery data. For example:

   ```
   --download-dir=C:\Users\user1\Documents\FlightGear
   ```

   All data downloaded during this FlightGear session is saved to the specified directory. To avoid downloading duplicate scenery data, use the same directory in succeeding FlightGear sessions
3. To open FlightGear, run `runfg.bat`.

Note Each time that you run the Generate Run Script block, it creates a new script. It overwrites any edits that you have added.

See Also

FlightGear Preconfigured 6DoF Animation | Generate Run Script | Pack net_fdm Packet for FlightGear | Receive net_ctrl Packet from FlightGear | Send net_fdm Packet to FlightGear | Unpack net_ctrl Packet from FlightGear

Related Examples

- “Work with the Flight Simulator Interface” on page 2-24

External Websites

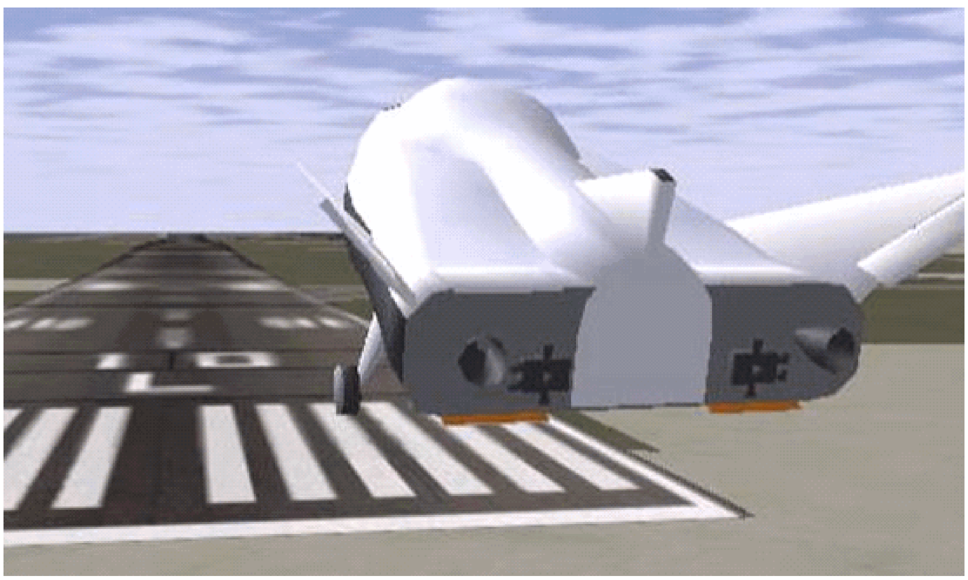
- https://www.flightgear.org
- Hardware recommendations section of the FlightGear website

Work with the Flight Simulator Interface

In this section...

Introduction

Use this section to learn how to use the FlightGear flight simulator and the Aerospace Blockset software to visualize your Simulink aircraft models. If you have not yet installed FlightGear, see "Flight Simulator Interface" on page 2-20 first.



Simulink Driven HL-20 Model in a Landing Flare at KSFC

About Aircraft Geometry Models

Before you can visualize your aircraft dynamics, you need to create or obtain an aircraft model file compatible with FlightGear. This section explains how to do this.

Aircraft Geometry Editors and Formats

You have a competitive choice of over twelve 3-D geometry file formats supported by FlightGear.

Currently, the most popular 3-D geometry file format is the AC3D format, which has the suffix `*.ac`. AC3D is a low-cost geometry editor available from `https://www.inivis.com`.

Aircraft Model Structure and Requirements

Aircraft models are contained in the `FlightGearRoot/data/Aircraft/` folder and subfolders. A complete aircraft model must contain a folder linked through the required file named `model-set.xml`.

All other model elements are optional. This is a partial list of the optional elements you can put in an aircraft data folder:

- Vehicle objects and their shapes and colors
- Vehicle objects' surface bitmaps
- Variable geometry descriptions
- Cockpit instrument 3-D models
- Vehicle sounds to tie to events (e.g., engine, gear, wind noise)
- Flight dynamics model
- Simulator views
- Submodels (independently movable items) associated with the vehicle

Model behavior reverts to defaults when these elements are not used. For example,

- Default sound: no vehicle-related sounds are emitted.
- Default instrument panel: no instruments are shown.

Models can contain some, all, or even none of the above elements. If you always run FlightGear from the cockpit view, the aircraft geometry is often secondary to the instrument geometries.

A how-to document for including optional elements is included in the FlightGear documentation at:

`https://wiki.flightgear.org/Howto:3D_Aircraft_Models`

Required Flight Dynamics Model Specification

The flight dynamics model (FDM) specification is a required element for an aircraft model. To set the Simulink software as the source of the flight dynamics model data stream for a given geometry model, you put this line in `data/Aircraft/model/model-set.xml`:

```
<flight-model>network</flight-model>
```

Obtain and Modify Existing Aircraft Models

You can quickly build models from scratch by referencing instruments, sounds, and other optional elements from existing FlightGear models. Such models provide examples of geometry, dynamics, instruments, views, and sounds. It is simple to copy an aircraft folder to a new name, rename the `model-set.xml file`, modify it for network flight dynamics, and then run FlightGear with the `—aircraft` flag set to the name in `model-set.xml`.

Many existing 3-D aircraft geometry models are available for use with FlightGear. Visit the download area of https://www.flightgear.org to see some of the aircraft models available. Additional models can be obtained via Web search. Search key words such as “flight gear aircraft model” are a good starting point. Be sure to comply with copyrights when distributing these files.

Hardware Requirements for Aircraft Geometry Rendering

When creating your own geometry files, keep in mind that your graphics card can efficiently render a limited number of surfaces. Some cards can efficiently render fewer than 1000 surfaces with bitmaps and specular reflections at the nominal rate of 30 frames per second. Other cards can easily render on the order of 10,000 surfaces.

If your performance slows while using a particular geometry, gauge the effect of geometric complexity on graphics performance by varying the number of aircraft model surfaces. An easy way to check this is to replace the full aircraft geometry file with a simple shape, such as a single triangle, then test FlightGear with this simpler geometry. If a geometry file is too complex for smooth display, use a 3-D geometry editor to simplify your model by reducing the number of surfaces in the geometry.

Work with Aircraft Geometry Models

Once you have obtained, modified, or created an aircraft data file, you need to put it in the correct folder for FlightGear to access it.

Import Aircraft Models into FlightGear

To install a compatible model into FlightGear, use one of these procedures. Choose the one appropriate for your platform. This section assumes that you have read “Install and Start FlightGear” on page 2-21.

If your platform is Windows:

1. Go to your installed FlightGear folder. Open the `data` folder, then the `Aircraft` folder: `\FlightGear\data\Aircraft\`.
2. Make a subfolder *`model`*`\` here for your aircraft data.
3. Put *`model`*`-set.xml` in that subfolder, plus any other files needed.

   It is common practice to make subdirectories for the vehicle geometry files (`\model\`), instruments (`\instruments\`), and sounds (`\sounds\`).

If your platform is Linux:

1. Go to your installed FlightGear directory. Open the `data` directory, then the `Aircraft` directory: *`$FlightGearBaseDirectory`*`/data/Aircraft/`.
2. Make a subdirectory *`model`*`/` here for your aircraft data.
3. Put *`model`*`-set.xml` in that subdirectory, plus any other files needed.

   It is common practice to make subdirectories for the vehicle geometry files (`/model/`), instruments (`/instruments/`), and sounds (`/sounds/`).

If your platform is Mac:

1. Open a terminal.
2. Go to your installed FlightGear folder. Open the `data` folder, then the `Aircraft` folder:

   ```
   $FlightGearBaseDirectory/FlightGear.app/Contents/Resources/data/Aircraft/
   ```
3. Make a subfolder *`model`*`/` here for your aircraft data.
4. Put *`model`*`-set.xml` in that subfolder, plus any other files needed.

It is common practice to make subdirectories for the vehicle geometry files (`/model/`), instruments (`/instruments/`), and sounds (`/sounds/`).

Example: Animate Vehicle Geometries

This example illustrates how to prepare hinge line definitions for animated elements such as vehicle control surfaces and landing gear. To enable animation, each element must be a named entity in a geometry file. The resulting code forms part of the HL20 lifting body model presented in “Run the HL-20 Example with FlightGear” on page 2-32.

1. The standard body coordinates used in FlightGear geometry models form a right-handed system, rotated from the standard body coordinate system in Y by -180 degrees:

   - X = positive toward the back of the vehicle
   - Y = positive toward the right of the vehicle
   - Z = positive is up, e.g., wheels typically have the lowest Z values.

   See “About Aerospace Coordinate Systems” on page 2-8 for more details.
2. Find two points that lie on the desired named-object hinge line in body coordinates and write them down as XYZ triplets or put them into a MATLAB calculation like this:

   ```
   a = [2.98, 1.89, 0.53];
   b = [3.54, 2.75, 1.46];
   ```
3. Calculate the difference between the points:

   ```
   pdiff = b - a
   pdiff =
       0.5600    0.8600    0.9300
   ```
4. The hinge point is either of the points in step 2 (or the midpoint as shown here):

   ```
   mid = a + pdiff/2
   mid =
       3.2600    2.3200    0.9950
   ```
5. Put the hinge point into the animation scope in *`model`*`-set.xml`:

   ```
   <center>
       <x-m>3.26</x-m>
       <y-m>2.32</y-m>
       <z-m>1.00</z-m>
   </center>
   ```
6. Use the difference from step 3 to define the relative motion vector in the animation axis:

   ```
   <axis>
       <x>0.56</x>
       <y>0.86</y>
       <z>0.93</z>
   </axis>
   ```
7. Put these steps together to obtain the complete hinge line animation used in the HL20 example model:

   ```
   <animation>
       <type>rotate</type>
       <object-name>RightAileron</object-name>
       <property>/surface-positions/right-aileron-pos-norm</property>
       <factor>30</factor>
       <offset-deg>0</offset-deg>
   ```

```
        <center>
            <x-m>3.26</x-m>
            <y-m>2.32</y-m>
            <z-m>1.00</z-m>
        </center>
        <axis>
            <x>0.56</x>
            <y>0.86</y>
            <z>0.93</z>
        </axis>
    </animation>
```

Run FlightGear with Simulink Models

To run a Simulink model of your aircraft and simultaneously animate it in FlightGear with an aircraft data file `model-set.xml`, you need to configure the aircraft data file and modify your Simulink model with some new blocks.

These are the main steps to connecting and using FlightGear with the Simulink software:

- “Set the Flight Dynamics Model to Network in the Aircraft Data File” on page 2-28 explains how to create the network connection you need.
- “Obtain the Destination IP Address” on page 2-28 starts by determining the IP address of the computer running FlightGear.
- “Send Simulink Data to FlightGear” on page 2-34 shows how to add and connect interface and pace blocks to your Simulink model.
- “Create a FlightGear Run Script” on page 2-29 shows how to write a FlightGear run script compatible with your Simulink model.
- “Start FlightGear” on page 2-30 guides you through the final steps to making the Simulink software work with FlightGear.
- “Improve Performance” on page 2-31 helps you speed your model up.
- “Run FlightGear and Simulink Software on Different Computers” on page 2-32 explains how to connect a simulation from the Simulink software running on one computer to FlightGear running on another computer.

Set the Flight Dynamics Model to Network in the Aircraft Data File

Be sure to:

- Remove any pre-existing flight dynamics model (FDM) data from the aircraft data file.
- Indicate in the aircraft data file that its FDM is streaming from the network by adding this line:

```
<flight-model>network</flight-model>
```

Obtain the Destination IP Address

You need the destination IP address for your Simulink model to stream its flight data to FlightGear.

- If you know your computer name, enter at the MATLAB command line:

```
java.net.InetAddress.getByName('www.mathworks.com')
```

- If you are running FlightGear and the Simulink software on the same computer, get your computer name by entering at the MATLAB command line:

```
java.net.InetAddress.getLocalHost
```

- If you are working in Windows, get your computer IP address by entering at the DOS prompt:

  ```
  ipconfig /all
  ```

  Examine the IP address entry in the resulting output. There is one entry per Ethernet device.

Create a FlightGear Run Script

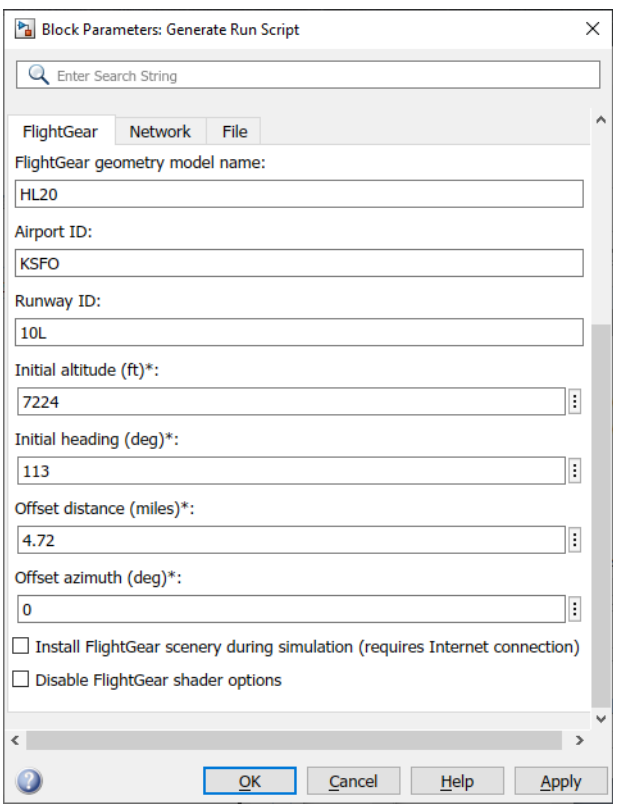
To start FlightGear with the desired initial conditions (location, date, time, weather, operating modes), it is best to create a run script by “Use the Generate Run Script Block” on page 2-29 or “Use the Interface Provided with FlightGear” on page 2-30.

If you make separate run scripts for each model you intend to link to FlightGear and place them in separate directories, run the appropriate script from the MATLAB interface just before starting your Simulink model.

Use the Generate Run Script Block

The easiest way to create a run script is by using the Generate Run Script block. Use this procedure:

1. Open the Flight Simulator Interfaces sublibrary.
2. Create a new Simulink model or open an existing model.
3. Drag a Generate Run Script block into the Simulink diagram.
4. Double-click the Generate Run Script block. Its dialog opens. Observe the three panes, **FlightGear**, **Network**, and **File**.



5. In the **Output file name** parameter of the **File** tab, type the name of the output file. This name should be the name of the command you want to use to start FlightGear with these initial parameters. Use the appropriate file extension:

Platform	Extension
Windows	`.bat`
Linux and macOS	`.sh`

   For example, if your file name is `runfg.bat`, use the `runfg` command to execute the run script and start FlightGear.
6. In the **FlightGear base directory** parameter of the **File** tab, specify the name of your FlightGear installation folder.
7. In the **FlightGear geometry model name** parameter of the **File** tab, specify the name of the subfolder, in the *`FlightGear`*`/data/Aircraft` folder, containing the desired model geometry.
8. Specify the initial conditions as needed.
9. Click the **Generate Script** button at the top of the **Parameters** area.

   The Aerospace Blockset software generates the run script, and saves it in your MATLAB working folder under the file name that you specified in the **File > Output file name** field.
10. Select or clear these check boxes and

    - To direct FlightGear to automatically install required scenery while the simulator is running — Select **Install FlightGear scenery during simulation (requires Internet connection)**. For Windows systems, you may encounter an error message while launching FlightGear with this option enabled. For more information, see “Install Additional FlightGear Scenery” on page 2-22.
    - To disable FlightGear shader options — Select **Disable FlightGear shader options**.
11. Repeat steps 5 through 10 to generate other run scripts, if needed.
12. Click **OK** to close the dialog box. You do not need to save the Generate Run Script block with the Simulink model.

The Generate Run Script block saves the run script as a text file in your working folder. This is an example of the contents of a run script file:

```
>> cd C:\Applications\FlightGear-<your_FlightGear_version>
>> SET FG_ROOT=C:\Applications\FlightGear-<your_FlightGear_version>\data
>> cd \bin\
>> fgfs --aircraft=HL20 --fdm=network,localhost,5501,5502,5503
  --fog-fastest --disable-clouds --start-date-lat=2004:06:01:09:00:00
  --disable-sound --in-air --enable-freeze --airport=KSFO --runway=10L
  --altitude=7224 --heading=113 --offset-distance=4.72 --offset-azimuth=0
```

Use the Interface Provided with FlightGear

The FlightGear launcher GUI (part of FlightGear, not the Aerospace Blockset product) lets you build simple and advanced options into a visible FlightGear run command.

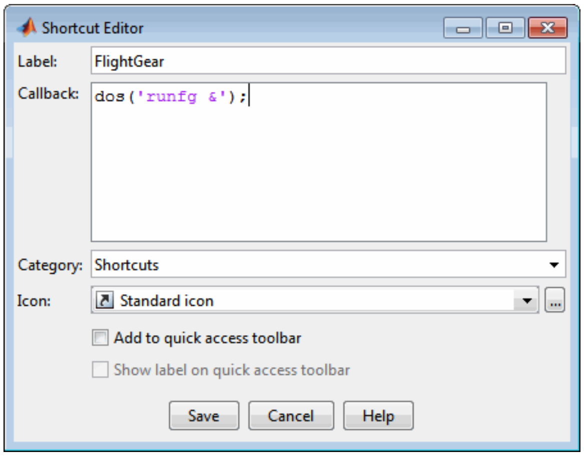
Start FlightGear

If your computer has enough computational power to run both the Simulink software and FlightGear at the same time, a simple way to start FlightGear on a Windows system is to create a MATLAB desktop button containing this command to execute a run script like the one created above:

```
system('runfg &')
```

To create a desktop button:

1 In the MATLAB Command Window, select **Shortcuts** and click **New Shortcut**. The **Shortcut Editor** dialog opens.
2 Set the **Label**, **Callback**, **Category**, and **Icon** fields as shown in this figure.



3 Click **Save**.

The **FlightGear** button appears in your MATLAB desktop. If you click it, the output file, for example `runfg.bat`, runs in the current folder.

Once you have completed the setup, start FlightGear and run your model:

1 Make sure your model is in a writable folder. Open the model, and update the diagram. This step ensures that any referenced block code is compiled and that the block diagram is compiled before running. Once you start FlightGear, it uses all available processor power while it is running.
2 Click the **FlightGear** button or run the FlightGear run script manually.
3 When FlightGear starts, it displays the initial view at the initial coordinates specified in the run script. If you are running the Simulink software and FlightGear on different computers, arrange to view the two displays at the same time.
4 Now begin the simulation and view the animation in FlightGear.

Improve Performance

If your Simulink model is complex and cannot run at the aggregate rate needed for the visualization, you might need to

- Use the Accelerator mode in Simulink (“Perform Acceleration”.)
- Free up processor power by running the Simulink model on one computer and FlightGear on another computer. Use the **Destination IP Address** parameter of the Send net_fdm Packet to FlightGear block to specify the network address of the computer where FlightGear is running.
- Simulate the Simulink model first, then save the resulting translations (*x*-axis, *y*-axis, *z*-axis) and positions (latitude, longitude, altitude), and use the FlightGear Animation object in Aerospace Toolbox to visualize this data.

Tip If FlightGear uses more computer resources than you want, you can change its scheduling priority to a lesser one. For example, see commands like Windows `start` and Linux `nice` or their equivalents.

Run FlightGear and Simulink Software on Different Computers

It is possible to simulate an aerospace system in the Simulink environment on one computer (the source) and use its simulation output to animate FlightGear on another computer (the target). The steps are similar to those already explained, with certain modifications.

1. Obtain the IP address of the computer running FlightGear. See "Obtain the Destination IP Address" on page 2-28.
2. Enter this target computer IP address in the Send net_fdm Packet to FlightGear block. See "Send Simulink Data to FlightGear" on page 2-34.
3. Update the Generate Run Script block in your model with the target computer FlightGear base folder. Regenerate the run script to reflect the target computer separate identity.

   See "Create a FlightGear Run Script" on page 2-29.
4. Copy the generated run script to the target computer. Start FlightGear there. See "Start FlightGear" on page 2-30.
5. If you want to also receive data from FlightGear, use the Receive net_ctrl Packet from FlightGear block. Enter the IP address of the computer running FlightGear in the **Origin IP address** parameter.
6. Update the run script for the receive data. Use the Generate Run Script block to regenerate the run script.
7. Start your Simulink model on the source computer. FlightGear running on the target displays the simulation motion.

Run the HL-20 Example with FlightGear

The Aerospace Blockset software contains an example model of the HL-20 lifting body that uses the FlightGear interface and projects. This example illustrates many features of the Aerospace Blockset software. It also contains a Variant Subsystem block that you can use to specify the data source for the simulation. You might want to use the Variant Subsystem block to change the terrain data source or if you do not want to use FlightGear but still want to simulate the model.

To install and configure FlightGear before attempting to simulate this model, see "Flight Simulator Interface" on page 2-20. Also, before attempting to simulate this model, read "Install and Start FlightGear" on page 2-21.

Note Step 2 of this example copies the preconfigured geometries for the HL-20 simulation from *`projectroot`*`\support` to *`FlightGear`*`\data\Aircraft\`. It requires that you have system administrator privileges for your machine. If you do not have these privileges, manually copy these files, depending on your platform.

Windows

Copy the `HL20` folder from *`projectroot`*`\support` folder to *`FlightGear`*`\data\Aircraft\` folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file *`projectroot`*`\support\HL20\Models\HL20.xml` defines the geometry.

For Windows platforms, start the MATLAB app with administrator privileges. For example, in the Start menu, right click the MATLAB app, then select **Run as administrator**.

For more information, see “Import Aircraft Models into FlightGear” on page 2-26.

Linux

Copy the `HL20` directory from *`projectroot`*`/support` directory to *`$FlightGearBaseDirectory`*`/data/Aircraft/` directory. This directory contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file *`projectroot`*`/support/HL20/Models/HL20.xml` defines the geometry.

For more about this step, see “Import Aircraft Models into FlightGear” on page 2-26.

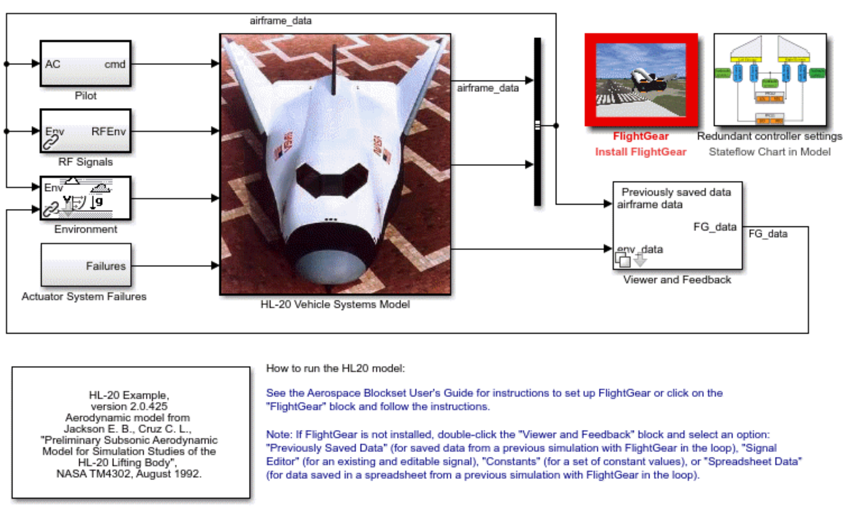
Mac

Copy the `HL20` folder from *`projectroot`*`/support` folder to *`$FlightGearBaseDirectory`*`/FlightGear.app/Contents/Resources/data/Aircraft/` folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file *`projectroot`*`/support/HL20/Models/HL20.xml` defines the geometry.

For more about this step, see “Import Aircraft Models into FlightGear” on page 2-26.

---

1. Start the MATLAB interface. Open the “HL-20 Project with Optional FlightGear Interface” on page 9-53example. The project for the model starts and the model opens.



2. If this is your first time running FlightGear for this model, you need to create and run a customized FlightGear run script. You can do this with one of these:

   - In the model, double-click the Install FlightGear block and follow the steps in the block. Initially, this block is red. As you follow the steps outlined in the block, the block mask changes.

     To start FlightGear for the model, click **Launch HL20 in FlightGear**.

3 Now start the simulation and view the animation in FlightGear.

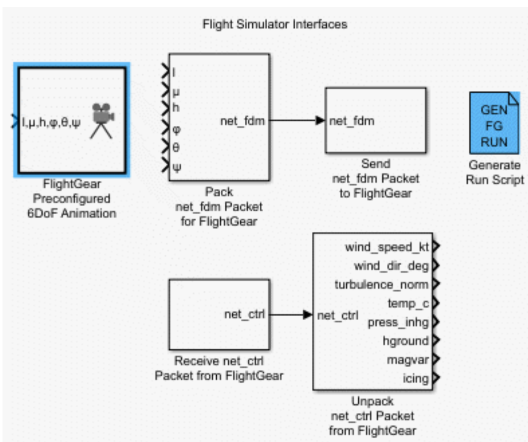
Note With the FlightGear window in focus, press the **V** key to alternate between the different aircraft views: cockpit view, helicopter view, chase view, and so on.

Send and Receive Data

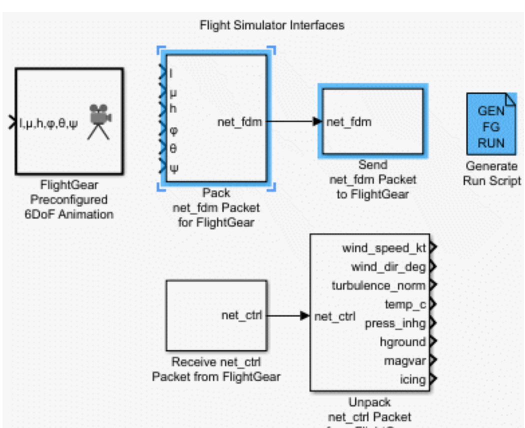
You can send and receive data between a Simulink model and a running FlightGear Flight Simulator.

Send Simulink Data to FlightGear

The easiest way to connect your model to FlightGear with the blockset is to use the FlightGear Preconfigured 6DoF Animation block:

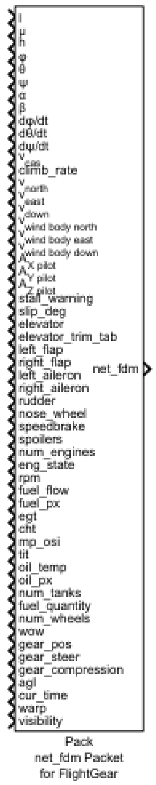


The FlightGear Preconfigured 6DoF Animation block is a subsystem containing the Pack net_fdm Packet for FlightGear and Send net_fdm Packet to FlightGear blocks:



These blocks transmit data from a model to a FlightGear session. The blocks are separate for maximum flexibility and compatibility.

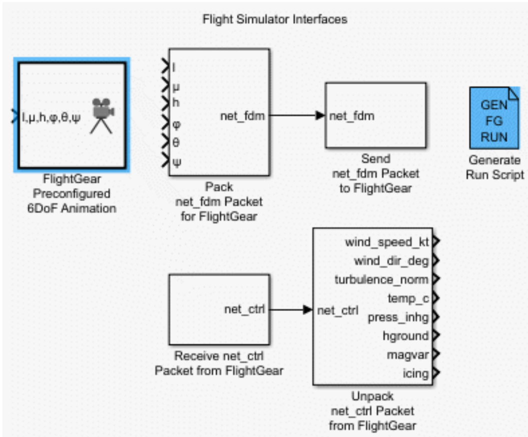
- The Pack net_fdm Packet for FlightGear block formats a binary structure compatible with FlightGear from model inputs. In the default configuration, the block displays only the 6DoF ports, but you can configure the full FlightGear interface supporting more than 50 distinct signals from the block dialog box:



- The Send net_fdm Packet to FlightGear block transmits this packet via UDP to the specified IP address and port where a FlightGear session awaits an incoming datastream. Use the IP address you found in “Obtain the Destination IP Address” on page 2-28.
- The Simulation Pace block slows the simulation so that its aggregate run rate is 1 second of simulation time per second of clock time. You can also use it to specify other ratios of simulation time to clock time.

Send FlightGear Data to Model

To increase the accuracy of your model simulation, you might want to send FlightGear environment variables to the Simulink model. Use these blocks:



- Receive net_ctrl Packet from FlightGear — Receives a network control and environment data packet `net_ctrl` from either the simulation of a Simulink model in the FlightGear simulator, or from a FlightGear session.
- Unpack net_ctrl Packet from FlightGear — Unpacks net_ctrl variable packets received from FlightGear and makes them available for the Simulink environment.
- Generate Run Script — Generates a customized FlightGear run script on the current platform.

For an example of how to use these blocks to send data to a Simulink model, see “HL-20 Project with Optional FlightGear Interface” on page 9-53.

These blocks use UDP to transfer data from FlightGear to the Simulink environment. Note:

- When a host and target are Windows or Linux platforms, you can use any combination of Windows or Linux platforms for the host and target.
- When a host or target is a Mac platform, use only Mac platforms for both the host and target.

See Also

FlightGear Preconfigured 6DoF Animation | Generate Run Script | Pack net_fdm Packet for FlightGear | Receive net_ctrl Packet from FlightGear | Send net_fdm Packet to FlightGear | Unpack net_ctrl Packet from FlightGear

Related Examples

- “Flight Simulator Interface” on page 2-20

External Websites

- https://www.flightgear.org

Unreal Engine Simulation Environment Requirements and Limitations

Aerospace Blockset provides an interface to a simulation environment that is visualized using the Unreal Engine from Epic Games®. When simulating in this environment, keep these requirements and limitations in mind.

Software Requirements

Note Simulating models in the 3D visualization environment requires Simulink 3D Animation.

The table summarizes the software requirements for Windows and Linux platforms.

Software	Windows	Linux
Operating system	Windows 64-bit	Debian® 11 (*recommended*)
Integrated development environment (IDE)	Visual Studio®	Visual Studio Code

Graphics driver	Microsoft® DirectX® — If this software is not already installed on your machine and you try to simulate in the environment, the toolbox prompts you to install it. Once you install the software, you must restart the simulation.	Vulkan®: AMD® (21.11.3+) and NVIDIA (515.48+) (*recommended*)
		
		Vulkan — If this software is not already installed on your machine and you try to simulate in the environment, the toolbox returns an error. To install the software, run these commands in the Linux terminal.
		
		`sudo apt install vulkan-tools`
		`sudo apt install mesa-vulkan-drivers`
		`sudo apt install nvidia-vulkan-common`
		
		To verify if the software is installed, run this command.
		
		`vkcube`
		
		Once you install the software, you must restart the simulation.

If you are customizing scenes, you must install Unreal Editor. In addition, verify that your Visual Studio and Unreal Engine project versions are supported by your MATLAB release.

MATLAB Release	Unreal Engine Version	Visual Studio Version
R2021b	4.25	2019
R2022a–R2022b	4.26	2019
R2023a–R2023b	4.27	2019

MATLAB Release	Unreal Engine Version	Visual Studio Version
R2024a	5.1	2022

Recommended Hardware Requirements

The table summarizes the recommended hardware requirements for Windows and Linux platforms.

Hardware	Windows	Linux
Graphics card (GPU)	Virtual reality-ready	NVIDIA GeForce® 960 GTX or higher with latest binary drivers
Video memory (RAM)	8 GB	8 GB
Processor (CPU)	2.5 GHz or faster	2.5 GHz or faster
Processor memory (RAM)	32 GB	32 GB

Limitations

The 3D simulation engine is not supported on Mac platforms.

The Simulation 3D blocks do not support:

- Code generation
- Model reference
- Multiple instances of the Simulation 3D Scene Configuration block
- Multiple Unreal Engine instances in the same MATLAB session
- Rapid accelerator mode
- Simulink Online™ simulation.

You cannot create or use `sim3d` objects or functions in MATLAB Online.

In addition, when using these blocks in a closed-loop simulation, all Simulation 3D blocks must be in the same subsystem.

See Also

More About

- “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2

External Websites

- Unreal Engine Documentation

Acknowledgements

Aerospace Blockset uses the Unreal® Engine. Unreal® is a trademark or registered trademark of Epic Games, Inc. in the United States of America and elsewhere.

How 3D Simulation for Aerospace Blockset Works

The aerospace models run programmable maneuvers in a photorealistic 3D visualization environment. Aerospace Blockset integrates the 3D simulation environment with Simulink so that you can query the world around aerospace vehicles for virtually testing perception, control, and planning algorithms. The Aerospace Blockset visualization environment uses the Unreal Engine by Epic Games.

Understanding how this simulation environment works can help you troubleshoot issues and customize your models.

Communication with 3D Simulation Environment

When you use Aerospace Blockset to run your algorithms, Simulink co-simulates the algorithms in the visualization engine.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

In the Simulink environment, Aerospace Blockset:

- Determines the next position of objects by using 3D visualization environment feedback and aerospace vehicle dynamics models.
- Configures the 3D visualization environment, specifically:
  - Ray tracing
  - Scene capture cameras
  - Initial object positions
- In the visualization engine environment, Aerospace Blockset positions the objects and uses ray tracing to query the environment.

Block Execution Order

During simulation, the 3D simulation blocks follow a specific execution order:

1. The aerospace blocks initialize the vehicles and send their **X**, **Y**, and **Yaw** signal data to the Simulation 3D Scene Configuration block.
2. The Simulation 3D Scene Configuration block receives the vehicle data and sends it to the sensor blocks.
3. The sensor blocks receive the vehicle data and use it to accurately locate and visualize the vehicles.

The **Priority** property of the blocks controls this execution order. To access this property for any block, right-click the block, select **Properties**, and click the **General** tab. By default, Simulation 3D Aircraft blocks have a priority of `-1`, Simulation 3D Scene Configuration blocks have a priority of `0`, and sensor blocks have a priority of `1`.

If your sensors are not detecting vehicles in the scene, it is possible that the 3D simulation blocks are executing out of order. Try updating the execution order and simulating again. For more details on execution order, see “Control and Display Execution Order”.

Also be sure that all 3D simulation blocks are located in the same subsystem. Even if the blocks have the correct **Priority** settings, if they are located in different subsystems, they still might execute out of order.

See Also

Related Examples

- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37
- “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2
- “Customize Scenes Using Simulink and Unreal Editor” on page 4-7
- “Get Started Communicating with the Unreal Engine Visualization Environment” (Vehicle Dynamics Blockset)
- “Prepare Custom Vehicle Mesh for the Unreal Editor” (Vehicle Dynamics Blockset)
- “Place Cameras on Actors in the Unreal Editor” (Vehicle Dynamics Blockset)

External Websites

- Unreal Engine

Visualize with Cesium

Aerospace Blockset can now stream 3D map and terrain data of a location using Cesium Ion®. Aerospace Blockset uses Unreal Engine to build an executable window that visualizes your simulation and uses Cesium® to create a scene for that visualization at runtime. To customize scenes, download the Cesium for Unreal plugin and use the Aerospace Blockset Interface for Unreal Engine Projects. For information on installing Cesium for Unreal, see "Install Cesium for Unreal Plugin" on page 4-13. For more information on customizing scenes, see "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

Note

- To visualize with Cesium, you must start MATLAB as administrator.
- The folder installed with your product must be writeable.

  matlabroot`\toolbox\shared\sim3d_projects\automotive_project\UE4\WindowsNoEditor\Engine`

This example has you create a Cesium Ion, then create a model and configure it for simulation.

Set Up a Cesium Ion Account

1. Create or log into your Cesium Ion account at https://ion.cesium.com/.
2. On the Cesium Ion page, click **Access Tokens**.
3. On the **Access Tokens** page, click **Create token** to create a new access token. Enter a unique name and note it. Leave the other settings at their default values.

   You use this access token ID in the **Access token ID** parameter of the Simulation 3D Scene Configuration block. For more information on creating access tokens, see https://cesium.com/learn/ion/cesium-ion-access-tokens/.
4. Click **Create**.

To create a model to visualize a simulation using Cesium Ion, see "Create a Model for Visualization" on page 2-42.

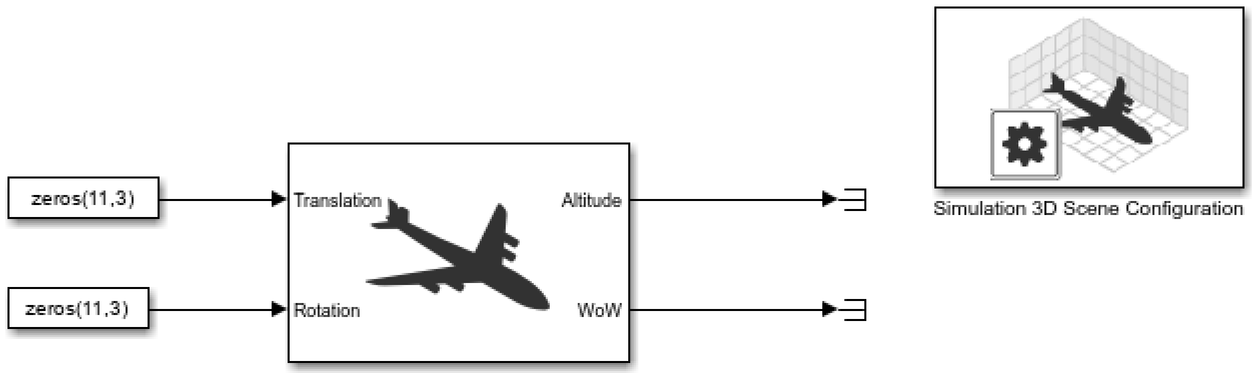
Create a Model for Visualization

1. Create a Simulink model and add these blocks:

   - Simulation 3D Aircraft block
   - Simulation 3D Scene Configuration block
   - Two Constant blocks
   - Two Terminator blocks
2. Open the Simulation 3D Aircraft block and set **Color** to `Green`.
3. Connect one Constant block to the **Translation** input port of the Simulation 3D Aircraft block. Set **Constant value** to `zeros(11,3)`.

4 Connect the second Constant block to the **Rotation** input port of the Simulation 3D Aircraft block. Set **Constant value** to `zeros(11,3)`.

5 Connect the **Attitude** and **WoW** ports of the Simulation 3D Aircraft block to the Terminator blocks.



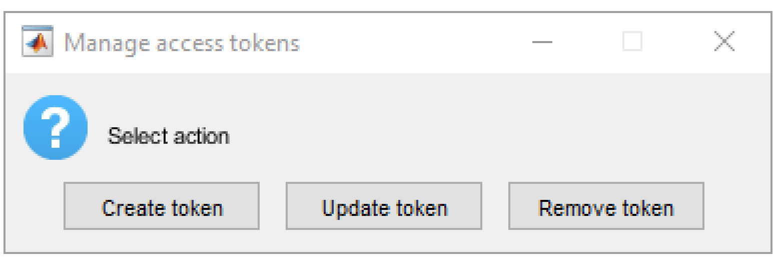
6 Set the model **Stop time** to `inf`.

Configure the Geospatial Configuration Parameters and Run the Simulation

1 In the model created from “Create a Model for Visualization” on page 2-42, open the Simulation 3D Scene Configuration block:

   a In the **Geospatial** tab, select **Enable geospatial configuration** to enable the geospatial configuration parameters.

   To configure this block for geospatial configuration, click the **Authentication Manager** button at the bottom of the pane. The **Manage access tokens** dialog box displays. Click **Create token**.

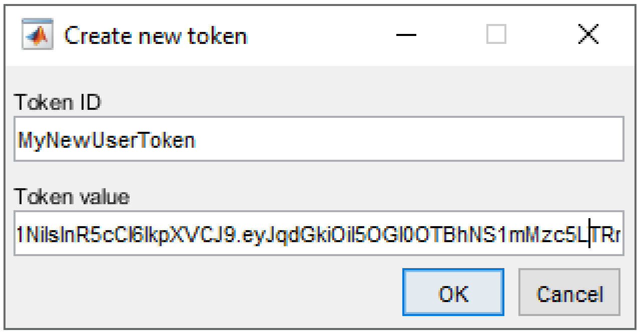


   The **Create new token** dialog box appears.

   b In the **Create new token** dialog box, enter the unique token ID string and copy and paste the access token created from the Cesium Ion page.

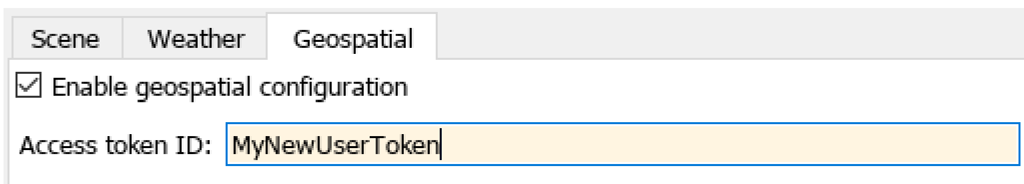
Copy From Cesium Ion	Paste To Authentication Manager > Create new token Dialog Box
Name	**Token ID**
Token	**Token value**

   For example, for token ID `MyNewUserToken`, enter these values.



c To store the token ID and value, click **OK**.

d In the **Access token ID** parameter of the **Geospatial** tab, enter the same unique token ID string you specified in the **Token ID** parameter of the **Authentication Manager > Create new token** dialog box.



e In the rest of the **Geospatial** tab, set these values.

Parameter	Value
Origin height	`3000`
Origin latitude	`41.892469`
Origin longitude	`-87.643373`
Map style	`Aerial`
Additional asset IDs	`[96188]`
Use advanced Sun Sky	Selected
Solar time	`8.25`
Time zone	`-5`
Day	`21`
Month	`9`
Year	`2022`
Use daylight saving time	Selected
DST start day	`10`
DST start month	`3`
DST end day	`3`
DST end month	`11`
DST switch hour	`2`

2 Run the simulation.

The simulation streams the aircraft with the 3D map and terrain data in an Unreal Engine executable window.

3 To view the aircraft from a preset view, left-click in the executable window and select a number from 1 to 9.

This view is from preset view 4.

See Also
Simulation 3D Aircraft | Simulation 3D Scene Configuration

Related Examples
- "Install Cesium for Unreal Plugin" on page 4-13
- "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2

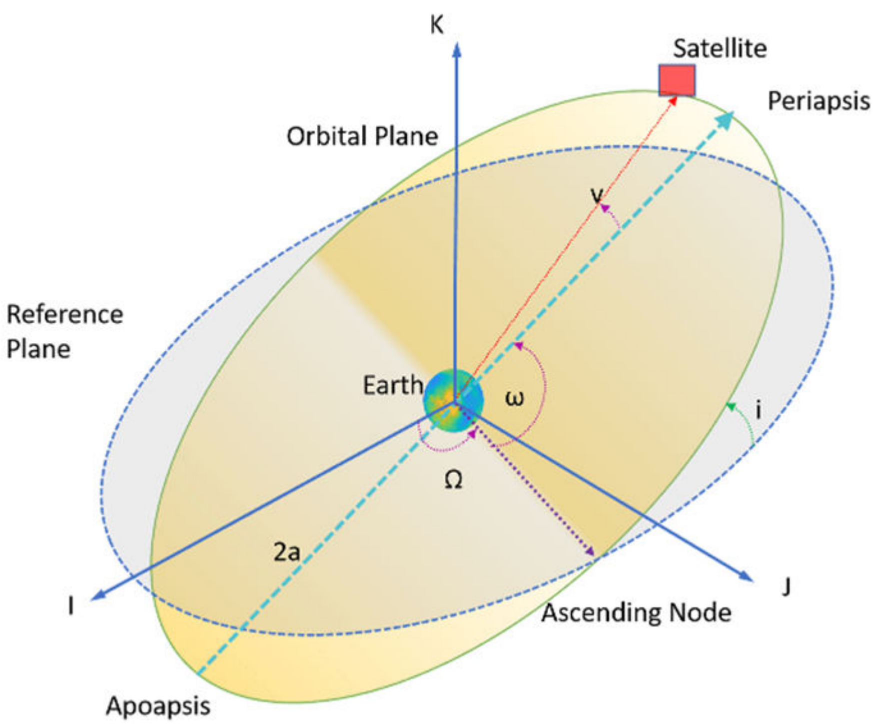
Orbit Propagation Methods

The Aerospace Blockset supports two top-level orbit propagation methods: `Kepler (unperturbed)` and `Numerical (high precision)`.

Kepler (unperturbed)

Kepler orbit propagation is the process of numerically computing and predicting the position and velocity of an object in space, based on Kepler's laws of planetary motion. Kepler's laws provide a mathematical description of the motion of objects in elliptical orbits around a central body, such as a planet orbiting the Sun.

To propagate a Keplerian orbit, you typically need these elements:



In this diagram, the orbital plane (yellow) intersects a reference plane (gray). For Earth-orbiting satellites, the reference plane is usually the **IJ**-plane of the GCRF.

These two elements define the shape and size of the ellipse:

- Eccentricity (e) — Describe how elongated the shape of the ellipse is when compared to a circle.
- Semimajor axis (a) — The sum of the periapsis and apoapsis distances divided by 2. Periapsis is the point at which an orbiting object is closest to the center of mass of the body it is orbiting. Apoapsis is the point at which an orbiting object is farthest away from the center of mass of the body it is orbiting. For classic two-body orbits, the semimajor axis is the distance between the centers of the bodies.

These two elements define the orientation of the orbital plane in which the ellipse is embedded:

- Inclination (i) — The vertical tilt of the ellipse with respect to the reference plane measured at the ascending node. The ascending node is the location where the orbit passes upward through the reference plane (the green angle i in the diagram). The tilt angle is measured perpendicular to the line of intersection between orbital plane and the reference plane. Any three points on an ellipse defines the ellipse orbital plane.

  Starting with an equatorial orbit, the orbital plane can be tilted up. The angle it is tilted up from the equator is referred to as the inclination angle, i, in the range [0,180]. Because the center of the Earth must always be in the orbital plane, the point in the orbit where the satellite passes the equator on its way up north-bound through the orbit is the ascending node, and the point where the satellite passes the equator on its way down south-bound is the descending node. Drawing a line through these two points on the equator defines the line of nodes.
- Right ascension of ascending node (Ω) — The horizontal orientation of the ascending node of the ellipse (where the orbit passes upward through the reference plane) with respect to the **I**-axis of the reference frame. The rotation of the right ascension of the ascending node (RAAN) can be any number between 0° and 360°.

The remaining two elements are:

- Argument of periapsis (ω) — The orientation of the ellipse in the orbital plane, as an angle measured from the ascending node to the periapsis in the range [0, 360).
- True Anomaly (ν) — The position of the orbiting body along the ellipse at a specific time. The position of the satellite on the path is measured counterclockwise from the periapsis and is called the true anomaly, ν, in the range [0, 360).

You can propagate the orbit using numerical methods such as numerical integration or Kepler's equation using these parameters. These methods allow you to calculate the position and velocity of the object at a given time. For information on each of these elements, see "Orbital Elements".

The Aerospace Blockset uses universal variables and Newton-Raphson iteration to propagate satellite orbits over time. This analytical algorithm is fast, but has limitations. Propagated orbits account only for central body spherical (point-mass) gravity. This formulation includes no other perturbations.

This propagation method is always performed in the ICRF inertial coordinate system with origin at the center of the central body. Given initial inertial position r_0 and velocity v_0 at time t_0, first find orbital energy, ξ, and the reciprocal of the semi-major axis, α:

$$\xi = \frac{v_0{}^2}{2} - \frac{\mu}{r_0}$$

$$\alpha = \frac{-2\xi}{\mu},$$

where μ is the standard gravitation parameter of the central body. Next, determine the orbit type from the sign of α.

- α>0 => Circular or elliptical
- α<0 => Hyperbolic
- α≈0 => Parabolic

To initialize the Newton-Raphson iteration, select an initial guess for χ based on the orbit type:

- Circular or elliptical orbit

  $$\chi_0 \approx \sqrt{\mu}(\Delta t)\alpha,$$

where Δt is the propagation step size (simulation time step). If Δt exceeds the orbital period $T = 2\pi\sqrt{\frac{a^3}{\mu}}$, wrap Δt.

- Parabolic orbit

$$\chi_0 \approx \sqrt{p}2\cot(2w),$$

where:

$$\vec{h} = \vec{r_0} \times \vec{v_0}$$

$$p = \frac{h \cdot h}{\mu}$$

$$\cot(2s) = 3\sqrt{\frac{\mu}{p^3}}(\Delta t)$$

$$\tan^3(w) = \tan(s)\,.$$

- Hyperbolic orbit:

$$\chi_0 \approx \mathrm{sign}(\Delta t)\sqrt{-\frac{1}{\alpha}}\ln\left(\frac{-2\mu\alpha(\Delta t)}{\vec{r_0} \cdot \vec{v_0} + \mathrm{sign}(\Delta t)\sqrt{-\frac{\mu}{\alpha}}(1 - r_0\alpha)}\right).$$

Perform Newton-Raphson iteration while $|x_n\text{-}x_{n-1}| > tolerance$.

$$\chi_{n+1} = \chi_n + \frac{\sqrt{\mu}(\Delta t) - \chi_n{}^3c_3 - \frac{\vec{r_0} \cdot \vec{v_0}}{\sqrt{\mu}}\chi_n{}^2c_2 - r_0\,\chi_n(1 - \psi c_3)}{\chi_n{}^2c_2 + \frac{\vec{r_0} \cdot \vec{v_0}}{\sqrt{\mu}}\chi_n(1 - \psi c_3) + r_0(1 - \psi c_2)}$$

$$\chi_n \Leftarrow \chi_{n+1},$$

where:

$$\psi = \chi_n{}^2\alpha\,.$$

(if ψ>0),

$$c_2 = \frac{1 - \cos(\sqrt{\psi})}{\psi}$$

$$c_3 = \frac{\sqrt{\psi} - \sin(\sqrt{\psi})}{\sqrt{\psi^3}}\,.$$

(if ψ<0),

$$c_2 = \frac{1 - \cosh(\sqrt{-\psi})}{\psi}$$

$$c_3 = \frac{\sinh(\sqrt{-\psi}) - \sqrt{-\psi}}{\sqrt{(-\psi)^3}}\,.$$

(if ψ≈0),

$$c_2 = \frac{1}{2}$$

$$c_3 = \frac{1}{6}\,.$$

Calculate universal variables f, $\dot{f}$, g, and $\dot{g}$.

$$f = 1 - \frac{\chi_n^2}{r_0} c_2$$

$$\dot{f} = \frac{\sqrt{\mu}}{r r_0} \chi_n (\psi c_3 - 1)$$

$$g = (\Delta t) - \frac{\chi_n^3}{\sqrt{\mu}} c_3$$

$$\dot{g} = 1 - \frac{\chi_n^2}{r} c_2 .$$

Assemble position and velocity output vectors:

$$\overrightarrow{r_{\text{icrf}}} = f \overrightarrow{r_0} + g \overrightarrow{v_0}$$

$$\overrightarrow{v_{\text{icrf}}} = \dot{f} \overrightarrow{r_0} + \dot{g} \overrightarrow{v_0} .$$

Numerical (high precision)

This option uses the Simulink solver to integrate position and velocity from central body gravitational acceleration at each simulation timestep (Δt). The method for computing central body acceleration depends on the current setting for the **Gravitational potential model** parameter. You can also include custom acceleration components in the propagation algorithm using the $\mathbf{A}_{\text{icrf}}$ (applied acceleration) input port. For gravity models that include nonspherical acceleration terms, the block computes nonspherical gravity in a fixed-frame coordinate system (ITRF, in the case of Earth). Numerical integration, however, is always performed in the inertial ICRF coordinate system. Therefore, at each timestep:

1. The block transforms position and velocity states into the fixed-frame.
2. The block calculates nonspherical gravity in the fixed-frame.
3. The block transforms resulting acceleration into the inertial frame, where it is summed with the other acceleration terms and integrated.

$$\overrightarrow{a_{\text{icrf}}} = \vec{a}_{\text{central body gravity}} + \vec{a}_{\text{applied}}$$

$$\overrightarrow{a_{\text{icrf}}} \xRightarrow[\text{integrate}]{} \overrightarrow{r_{\text{icrf}}}, \overrightarrow{v_{\text{icrf}}}$$

Central Body

Methods for computing central body acceleration include point-mass, oblate ellipsoid, or spherical harmonics.

- *Point-mass* (available for all central bodies) — This option treats the central body as a point-mass, including only the effects of spherical gravity using Newton's law of universal gravitation.

  $$\vec{a}_{\text{centralbodygravity}} = -\frac{\mu}{r^2} \frac{\overrightarrow{r_{\text{icrf}}}}{r},$$

  where μ is the standard gravitation parameter of the central body.
- *Oblate ellipsoid (J2)* (available for all central bodies) — In addition to spherical gravity, this option includes the perturbing effects of the second-degree, zonal harmonic gravity coefficient J_2,

accounting for the oblateness of the central body. J_2 accounts for the vast majority of the central bodies gravitational departure from a perfect sphere.

$$\vec{a}_{centralbodygravity} = -\frac{\mu}{r^2}\frac{\overrightarrow{r_{\mathrm{icrf}}}}{r} + fixed2inertial(\vec{a}_{nonspherical}),$$

where:

$$\vec{a}_{\mathrm{nonspherical}} =$$
$$\left\{[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{\mathrm{ff}_k}}{r^2\sqrt{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}}\frac{\partial}{\partial\phi}U]r_{\mathrm{ff}_i}\right\}i$$
$$+\left\{[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{\mathrm{ff}_k}}{r^2\sqrt{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}}\frac{\partial}{\partial\phi}U]r_{\mathrm{ff}_j}\right\}j$$
$$+\left\{\frac{1}{r}(\frac{\partial}{\partial r}U)r_k + \frac{\sqrt{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}}{r^2}\frac{\partial}{\partial\phi}U\right\}k,$$

given the partial derivatives in spherical coordinates:

$$\frac{\partial}{\partial r}U = \frac{3\mu}{r^2}(\frac{R_{\mathrm{cb}}}{r}P_{2,0}[\sin(\phi)]J_2$$
$$\frac{\partial}{\partial\phi}U = -\frac{\mu}{r}(\frac{R_{\mathrm{cb}}}{r}P_{2,1}[\sin(\phi)]J_2$$

where:

- ϕ and λ — Satellite geocentric latitude and longitude.
- $P_{2,0}$ and $P_{2,1}$ — Associated Legendre functions.
- μ — Standard gravitation parameter of the central body.
- R_{cb} — Central body equatorial radius.

The transformation `fixed2inertial` converts fixed-frame position, velocity, and acceleration into the ICRF coordinate system with origin at the center of the central body, accounting for centrifugal and coriolis acceleration. For more information about the fixed and inertial coordinate systems used for each central body, see “Coordinate Systems” on page 5-669.

- *Spherical Harmonics* (available for Earth, Moon, Mars, Custom) — This option adds increased fidelity by including higher-order perturbation effects accounting for zonal, sectoral, and tesseral harmonics. For reference, the second-degree, zeroth order zonal harmonic J_2=-C_{20}. The Spherical Harmonics model accounts for harmonics up to max degree l=l_{max}, which varies by central body and geopotential model.

$$\vec{a}_{centralbodygravity} = -\frac{\mu}{r^2}\frac{\overrightarrow{r_{\mathrm{icrf}}}}{r} + fixed2inertial(\vec{a}_{nonspherical}),$$

where:

$$\vec{a}_{nonspherical} =$$

$$\left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{\mathrm{ff}_k}}{r^2\sqrt{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{\mathrm{ff}_i} - \left[\frac{1}{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}\frac{\partial}{\partial \lambda}U\right]r_{\mathrm{ff}_j}\right\}i$$

$$+\left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{\mathrm{ff}_k}}{r^2\sqrt{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{\mathrm{ff}_j} + \left[\frac{1}{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}\frac{\partial}{\partial \lambda}U\right]r_{\mathrm{ff}_i}\right\}j$$

$$+\left\{\frac{1}{r}\left(\frac{\partial}{\partial r}U\right)r_{\mathrm{ff}_k} + \frac{\sqrt{r_{\mathrm{ff}_i}{}^2 + r_{\mathrm{ff}_j}{}^2}}{r^2}\frac{\partial}{\partial \phi}U\right\}k,$$

given the following partial derivatives in spherical coordinates:

$$\frac{\partial}{\partial r}U = -\frac{\mu}{r^2}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\mathrm{cb}}}{r}\right)(l+1)P_{l,m}[\sin(\phi)]\left\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\right\}$$

$$\frac{\partial}{\partial \phi}U = \frac{\mu}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\mathrm{cb}}}{r}\right)\left\{P_{l,m+1}[\sin(\phi)] - (m)\tan(\phi)\,P_{l,m}[\sin(\phi)]\right\}\left\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\right\}$$

$$\frac{\partial}{\partial \lambda}U = \frac{\mu}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\mathrm{cb}}}{r}\right)(m)P_{l,m}[\sin(\phi)]\left\{S_{l,m}\cos(m\lambda) - C_{l,m}\sin(m\lambda)\right\},$$

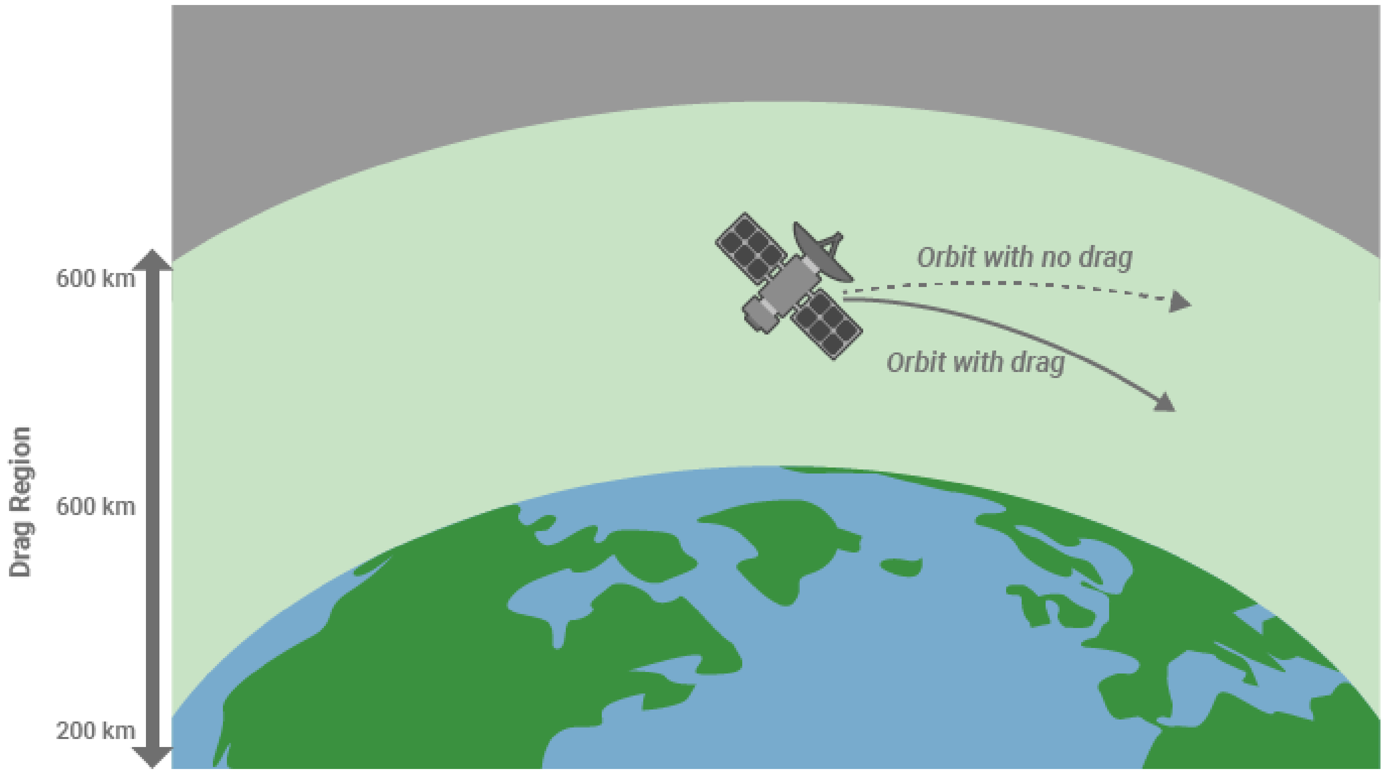
where:

- ϕ and λ — Satellite geocentric latitude and longitude.
- $P_{l,m}$ — Associated Legendre functions.
- μ — Standard gravitation parameter of the central body.
- R_{cb} — Central body equatorial radius.
- $C_{l,m}$ and $S_{l,m}$ — Nonnormalized harmonic coefficients.

The transformation `fixed2inertial` converts fixed-frame position, velocity, and acceleration into the ICRF coordinate system with origin at the center of the central body, accounting for centrifugal and coriolis acceleration. For more information about the fixed and inertial coordinate systems used for each central body, see “Coordinate Systems” on page 5-669.

Atmospheric Drag

Atmospheric drag is a force that opposes the motion of an object moving through a fluid, such as the atmosphere of the earth. Atmospheric drag is influenced by several factors, including the shape, size, and velocity of the object, and the properties of air. The primary mechanism behind atmospheric drag is the conversion of the kinetic energy of the object into heat through the process of air molecule collision and friction.



The Aerospace Blockset uses this atmospheric drag equation:

$$a_{drag} = -\frac{1}{2}\rho\left(\frac{C_D A}{m}\right)v^2{}_{rel}\frac{\vec{v}_{rel}}{v_{rel}}$$

where:

- m — Spacecraft mass used by atmospheric drag calculation.
- C_D — Coefficient of drag assuming that it is dimensionless.
- ρ — Atmospheric density.
- A — Area normal to v_{rel}, where

  $$\vec{v}_{rel} = \vec{v}_{sat} + \vec{v}_{atmos}$$
- v_{rel} — Velocity relative to atmosphere.

  $$\vec{v}_{rel} = \vec{v}_{icrf} - \vec{\omega}_{\oplus} \times \overrightarrow{r_{icrf}}$$

  where $\vec{\omega}_{\oplus}$ is the central body angular velocity.

Third Body

In Aerospace Blockset, third body refers to an additional celestial object that influences the motion of two primary bodies in a gravitational system.

This equation incorporates the third body contribution, enabling a more precise prediction of the motion of objects in space.

$$a_{third\_body} = \mu_{third}\left(\frac{\vec{r}_{sat,3}}{r^3_{sat,3}} - \frac{\vec{r}}{r^3}\right)$$

where:

- μ_{third} — Gravitational parameter of the third body.
- $\vec{r}_{sat,3}$ — Vector from the satellite to the third body.
- $\vec{r}$ — Position of third body with regard to the central body, specified as a vector.

Solar Radiation Pressure

Solar Radiation Pressure (SRP) is the force produced by the impact of sunlight photons on the surface of an object in space. SRP is considered as a perturbing force that needs to be accounted for in orbit determination and prediction.

The Aerospace Blockset uses this solar radiation pressure equation:

$$a_{srp} = -\upsilon C_r \frac{A_s}{m} P_{srp}\left(\frac{AU}{r_{sat,sun}}\right)^2 \frac{\vec{r}_{sat,sun}}{r_{sat,sun}}$$

where:

- m — Mass of the spacecraft
- ν — Eclipse shadow function
- C_r — Spacecraft coefficient of reflectivity
- A_s — Spacecraft solar radiation pressure area
- P_{srp} — Solar radiation pressure at a distance of AU from the Sun
- AU — Mean distance from the Sun to Earth (1AU)
- $|\vec{r}_{sat,sun}|$ — Vector from the satellite to the Sun origin

See Also

Blocks
Orbit Propagator

Projects Template for Flight Simulation Applications

Flight Simulation Applications

Use projects to help organize large flight simulation modeling projects and makes it easier to share projects with others. This template provides a framework for the collaborative development of a flight simulation application. You can customize this project structure for specific applications.

Note To successfully run this example, install a C/C++ compiler.

The Aerospace Blockset software supplies a projects template that you can use to create your own flight simulation application. This template uses variant subsystems, model variants, and referenced models to implement flight simulation application components such as:

- An airframe that contains a 6DOF equation of motion environment model and actuator dynamics
- An inertial measurement unit (IMU) sensor model
- A visualization subsystem oriented for FlightGear
- A model of the nonlinear dynamics of the airframe
- A model of the linear dynamics of the airframe

Download the Flight Simulation Template

1. From the Simulink Start Page, select **Flight Simulation**.
2. In the Create Project window, in **Name**, enter a project name, for example `FlightSimProj`.
3. In **Folder**, enter a project folder or browse to the folder to contain the project, for example `FlightSimFolder`.
4. Click **OK**.

   If the folder does not exist, the dialog prompts you to create it. Click **Yes**.

   The software compiles the project, populates the project folders, and opens the main model, `flightSimulation`. All models and supporting files are in place for you to customize for your flight simulation application.

Contents of the Project Template

The flight simulation project template contains these folders

- **mainModels**

  Contains the top-level simulation model, `flightSimulation`. This model opens on startup. This file contains the top-level blocks for the flight simulation environment. Simulink uses the Variant Subsystem, Model Variants, and Model blocks at this level to adapt to the different simulation conditions.

  - The aircraft airframe can vary between a nonlinear an linear approach.
  - The commands to the aircraft can vary between a Signal Editor block, a joystick or a variable from the workspace.

  - Sensors can vary between models that include sensor dynamics or feedthrough (no associated dynamics).
  - Environment values can vary between state-dependent values (the values of temperature, pressure and so on depend on local position, latitude, etc.) or constant values that do not depend on state values.
  - The Visualization subsystem provides hooks that let you work with the states. For example, you can visualize the states using FlightGear or they can be recorded in a variable in the workspace for further analysis. States can also be visualized using the Simulation Data Inspector.
- **libraries**

  Contains the libraries used by the models.
- **nonlinearAirframe**

  Contains a model of the nonlinear dynamics of the airframe.

  - A specific subsystem (AC model) that contains a placeholder for the dynamics of your aircraft model. The characteristics of this subsystem are:

    - Actuators and environment inputs. Actuators refer to generic signals that may affect the behavior of the aircraft (for example an electric signal in voltage that will change the position of the hydraulic actuator connected to a control surface such as an aileron).
    - Forces and moments outputs. Effective in the center of gravity of the aircraft in body axis.
  - A 6DOF Body Quaternion block that solves the differential equations of forces and moments to obtain the aircraft states.
- **linearAirframe**

  Contains the linear dynamics of the airframe and the model to obtain these linear dynamics. The example obtains these dynamics by linearizing the nonlinear model using the `trimLinearizeOpPoint` function and `trimNonlinearAirframe` model. This function uses “Simulink Control Design” software to perform the linearization. It performs linearization of the nonlinear model for a given set of known inputs and conditions. For further information regarding trim and linearization, see the Simulink Control Design™ documentation). The `trimLinearizeOpPoint` function stores the output in a MAT-file.
- **controller**

  Contains the models for the Flight Control System (FCS) and its design. These models contain referenced models for different controller architectures needed for the design of aircraft simulation.
- **src**

  Contains source code such as C code. For simulation, it also has two folders that contain S-functions for simulation. These S-functions map buses to vectors and vice versa for the linear airframe model. This mapping can be changed depending on the linearization scheme, and the set of inputs and outputs for the model. To edit the indices for the different signals, you can use the S-Function Builder block.
- **tasks**

  Contains scripts to run the model. These scripts do not run continuously during the simulation process.

The folder also contains the non-virtual bus definitions for the states, environment, and sensor buses. These definitions, set the signals and characteristics that different elements in the simulation environment use. This folder also contains the definitions for the variables used in the mask workspace for the Sensors, FlightGear, linearAirframe and nonlinearAirframe blocks. These utilities store parameter values in data structures. For example, if the nonlinear model uses a parameter for a Gain block, the stored variable in the structure is `Vehicle.Nonlinear.Gain.gainValue`, which points to the parameter.

- **tests**

  Contains a sample test harness:

  - The `linearTest` file contains the actual test point. This file compares a subset of the outputs of the linearized airframe model to the outputs of the nonlinear airframe for the specific trim condition.
  - The `runProjectTests` file runs all the available files classified as "Tests" in the project.
- **utilities**

  Contains project-specific maintenance task utilities, such as:

  - `projectPaths` - Lists the location of folders to be added to the MATLAB path.
  - `rebuildSFunction` - Rebuilds S-functions for `linearInputBus` and `linearOutputBus`.
  - `startVars` - Defines the variables that the simulation environment requires to be in the base workspace. This utility also controls variants using the `Variants` structure. This structure lets the example switch between the nonlinear and linear airframe from the workspace by changing `VSS_VEHICLE` from `1` (for the nonlinear model) to `0` (for the linear model). For more information on subsystem variants see Model.
- **work**

  Contains files generated from every run. These files derive from source files, such as the MEX-file that you build from S-function C code.

In Shortcuts, projects creates shortcuts for common tasks:

- **Initialize Variables** — Runs the `startVars` script, which initializes the variables to the base workspace.
- **Rebuild S-functions** —Rebuilds the S-functions in the `src` folder.
- **Run Project Tests** —Runs the test points, labeled **Tests**, for test files in the project.
- **Top Level Simulation Model** — Opens the `flightSimulation` model. It runs on project startup.

Template Labels

Provides file classification labels for automatic and componentization sorting. This utility adds template labels such as **Tests**, **Airframe Design**, **Flight Controller Design**, and **Calibration Data**.

Add Airframe Dynamics and Controller Algorithm to the Project

1. To open the `linearAirframe` model, in `flightSimulation` double-click the Airframe subsystem.
2. Double-click the Nonlinear subsystem.

3. In the AC model, add your airframe dynamics.
4. Save the model.

Add Controller Algorithm to the Project

1. To open the `flightControlSystem` model, in `flightSimulation`, double-click the FCS subsystem.
2. In the Controller subsystem, add your controller algorithm.
3. Save the model.

Other things to try:

- Simulate your model.
- Explore the **tests** folder for sample tests for your application.

See Also

Related Examples

- "Create a New Project Using Templates"

More About

- "Model a Quadcopter Based on Parrot Minidrones" on page 3-22

Flight Instrument Gauges

Use the blocks for flight instrument gauges to visualize navigation variables, such as altitude and heading. These blocks, located in the Flight Instruments library, represent standard cockpit instruments:

- Airspeed Indicator
- Altimeter
- Artificial Horizon
- Climb Rate Indicator
- Exhaust Gas Temperature (EGT) Indicator
- Heading Indicator
- Revolutions Per Minute (RPM) Indicator
- Turn Coordinator

See Also

Airspeed Indicator | Altimeter | Artificial Horizon | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Related Examples

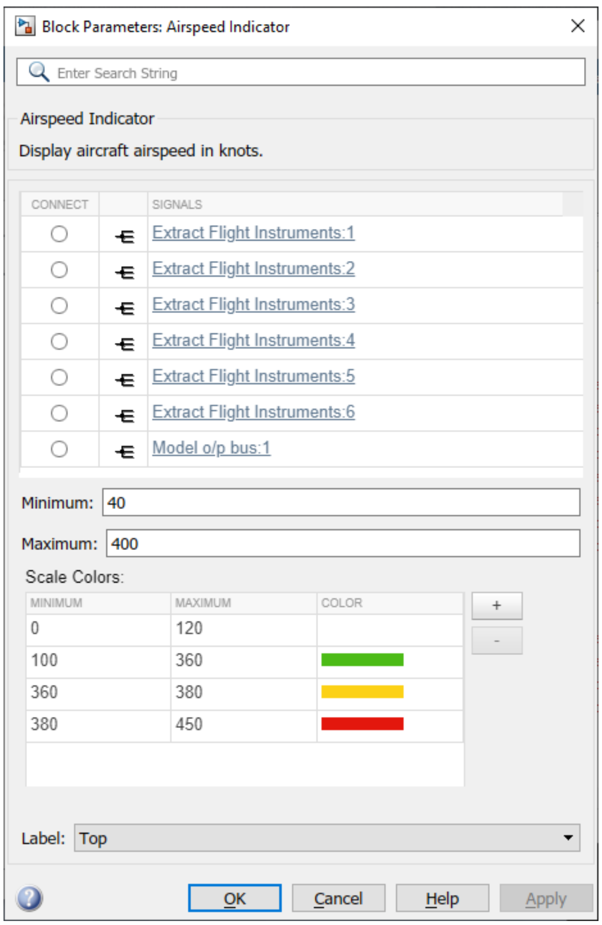
- “Display Measurements with Cockpit Instruments” on page 2-59
- “Programmatically Interact with Gauge Band Colors” on page 2-61

Display Measurements with Cockpit Instruments

You can view signal data using any of the flight instrument blocks. This example uses the "HL-20 with Flight Instrumentation Blocks" on page 9-36 model. In this example, connect a gauge so that you can view the aircraft heading.

1. To open the model, at the MATLAB command window, enter `openExample('HL20Gauges')`.
2. Open the Visualization subsystem.

   There is an existing Airspeed Indicator block in the model.
3. Add a second Airspeed Indicator block from the Flight Instruments library to the subsystem.
4. Open the new Airspeed Indicator block.
5. Select the Extract Flight Instruments block.
6. In the new Airspeed Indicator block, observe that the block connection table is filled with signals from the Extract Flight Instruments block that you can observe.



7. Select the option button next to `Extract_Gauges:2` in the connection table.
8. To connect the `Extract_Gauges:2` signal to the Airspeed Indicator block, click **OK**.

Tip To directly select the signal to connect, on the Extract Flight Instruments block, select the third output port (Roll Flightpath).

---

9 Simulate the model and observe the gauge as it registers the data.

10 To change the signal to connect to, you can:

- Select the same or another block and then select another signal in the updated block connection table.
- Select another output port for the same or a different block.

11 Close the model without saving it.

To create a Simulink model with prewired connections to flight instrument blocks, see flightControl3DOFAirframeTemplate.

See Also

Airspeed Indicator | Altimeter | Artificial Horizon | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

More About

- “Flight Instrument Gauges” on page 2-58
- “Programmatically Interact with Gauge Band Colors” on page 2-61

Programmatically Interact with Gauge Band Colors

You can programmatically change Airspeed Indicator, EGT Indicator, and RPM Indicator gauge band colors using the `ScaleColors` property. When used with `get_param`, this property returns an *n*-by-1 structure containing these elements, where *n* is the number of colored bands on the gauge:

- Min — Minimum value range for a color band
- Max — Maximum value range for a color band
- Color — RGB color triplet for a band (range from 0 to 1)

This example describes how to change a color band of the EGT Indicator gauge. By default, the EGT Indicator gauge looks like this.

This gauge has three bands, clockwise 1, 2, and 3.

1. Create a blank model and add an EGT Indicator block.
2. Select the EGT Indicator block.
3. To change the color bands for the EGT Indicator gauge, get the handle of the scale color objects.

```
sc=get_param(gcb,'ScaleColors')
```

```
sc =

  3×1 struct array with fields:

    Min
    Max
    Color
```

4. To see the values of the `Min`, `Max`, and `Color` values, use the `sc` handle. For example, to see the values of the first band, `sc(1)`, type:

```
sc(1)
```

```
sc(1)

ans =

  struct with fields:

      Min: 0
```

```
     Max: 700
   Color: [0.2980 0.7333 0.0902]
```

5 To change the color and size of this band, define a structure with different `Min`, `Max`, and `Color` values and set `ScaleColors` to that new structure. For example, to change the band range to 1 to 89 and the color to red:

```
sc(1) = struct('Min',1,'Max',89,'Color',[1 0 0]);
set_param(gcb,'ScaleColors',sc)
```

6 Observe the change in the EGT Indicator gauge.

7 You can add and change as many color bands as you need. For example, to add a fourth band and set up the gauge with that band:

```
sc(4) = struct('Min',200,'Max',300,'Color',[0 1 .6]);
set_param(gcb,'ScaleColors',sc)
```

See Also

Airspeed Indicator | Exhaust Gas Temperature (EGT) Indicator | Revolutions Per Minute (RPM) Indicator

More About

- “Flight Instrument Gauges” on page 2-58
- “Display Measurements with Cockpit Instruments” on page 2-59

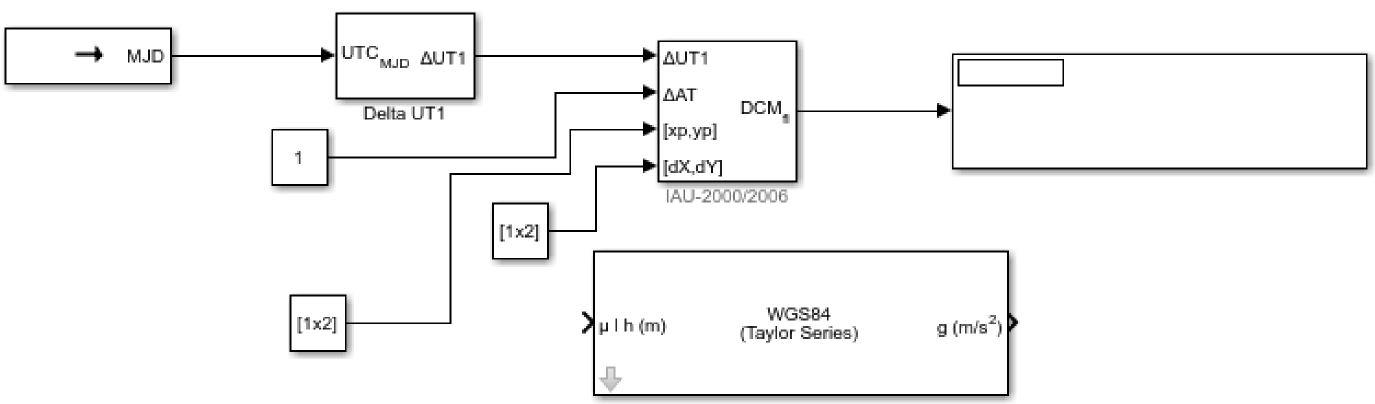
Calculate UT1 to UTC Values

Calculate the difference between principal Universal Time (UT1) and Coordinated Universal Time (UTC) according to International Earth Rotation Service (IERS) by using the Delta UT1 block. Use the Delta UT1 block with these axes transformation blocks:

- LLA to ECI Position
- ECI Position to LLA
- Direction Cosine Matrix ECI to ECEF
- ECI Position to AER

To calculate the difference between UT1 and UTC, the Delta UT1 block requires the modified Julian date. This example uses the Julian Date Conversion block. However, you can calculate the modified Julian data with other methods. For example, you can use the `mjuliandate` function from the Aerospace Toolbox software to calculate the date and input the result to the Delta UT1 block.

Use the Delta UT1 Block to Create Difference Values for the Direction Cosine Matrix ECI to ECEF Block



This model shows how a Direction Cosine Matrix ECI to ECEF block uses the output from the Delta UT1 and Julian Data Conversion blocks to obtain the difference between UTC and Universal Time (UT1).

1 Drag these blocks into a new model and connect them as shown:

   - Julian Date Conversion
   - Delta UT1
   - Direction Cosine Matrix ECI to ECEF
   - Display
   - Three Constant blocks

2 Set up the Julian Date Conversion to convert the date December 28, 2015 to its modified Julian date equivalent. This date must match the one specified in the Direction Cosine Matrix ECI to ECEF.

   - For **Year**, enter `2015`.

- For **Month**, enter 12.
- For **Day**, enter 28.
- To calculate the modified Julian date for December 28, 2015, select the **Calculate modified Julian date** check box.
- For **Time increment**, select None.

3 Leave the default settings for Delta UT1. By default, the block calculates the difference between Universal Time (UT1) and Coordinated Universal Time (UTC) to using the aeroiersdata.mat file supplied with the Aerospace Blockset software.

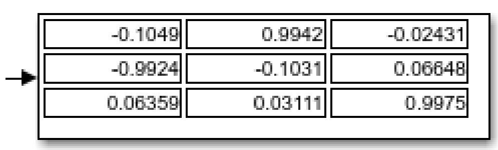
4 Set up the Direction Cosine Matrix ECI to ECEF block to work with the Coordinated Universal Time (UTC) December 28, 2015. This date must match the one specified in the Julian Date Conversion block:

- For **Year**, enter 2015.
- For **Month**, enter 12.
- For **Day**, enter 28.
- For **Time increment**, select None.

5 Set up the Δ*UT1*, Δ*AT*, and polar displacement of the Earth inputs for the Direction Cosine Matrix ECI to ECEF.

- Constant — Set **Constant value** to 1.
- Constant1 — Set **Constant value** to l.
- Constant2 — Set **Constant value** to [.05 .05].

6 Save and run the model. Observe the resulting direction cosine matrix in the Display block.



See Also

Delta UT1 | Direction Cosine Matrix ECI to ECEF | Julian Date Conversion

Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles

Aerospace Blockset provides flight control analysis tools that you can use to analyze the dynamic response and flying qualities of aerospace vehicles.

- “Flight Control Analysis Live Scripts” on page 2-65 — MATLAB live scripts demonstrate dynamic response and flying quality analysis of Sky Hogg and de Havilland Beaver airframes.
- “Modify Flight Control Analysis Templates” on page 2-67 — You can use templates to analyze the flying qualities of three degree-of-freedom and six degree-of-freedom airframe models. When you are comfortable running the analysis on the default airframes, you can replace them with your own airframe and analyze it.
- “Plot Short-Period Undamped Natural Frequency Results” on page 2-68 — After computing lateral-directional handling qualities, use the Aerospace Toolbox short-period functions to plot the short-period undamped natural frequency response.

Note Analyzing dynamic response and flying qualities of airframes requires a Simulink Control Design license.

Flight Control Analysis Live Scripts

Each flight control analysis template has an associated MATLAB live script that guides you through a flying quality analysis workflow for the default airframe. You can interact with the script and explore the analysis workflow.

- DehavillandBeaverFlyingQualityAnalysis — Compute longitudinal and lateral-directional flying qualities for a De Havilland Beaver airframe.
- SkyHoggLongitudinalFlyingQualityAnalysis — Compute longitudinal flying qualities for a Sky Hogg airframe.

For more information on running live scripts, see “Create and Run Sections in Code”.

1. Open one of the templates, for example:

   ```
   asbFlightControlAnalysis('6DOF')
   ```

   Navigate to the **Getting Started** section and click the first link.

   Alternatively, in the Command Window, type:

   ```
   open('DehavillandBeaverFlyingQualityAnalysis')
   ```
2. The script describes how to use eigenvalue analysis to determine the longitudinal flying qualities (long-period phugoid mode and short-period mode) and lateral-directional flying qualities (Dutch roll mode, roll mode, and spiral mode) for an aircraft modeled in Simulink.

   As you run the script, when applicable, the results of the runs display inline.

Modify Flight Analysis Templates

Aerospace Blockset provides these templates:

- flightControl6DOFAirframeTemplate — This template uses a six degree-of-freedom airframe configured for linearization and quality analysis. For initialization, the template uses the de Havilland Beaver airframe parameters.
- flightControl3DOFAirframeTemplate — This template uses a three degree-of-freedom longitudinal airframe configured for linearization and quality analysis. For initialization, the template uses Sky Hogg airframe parameters.

When you are comfortable navigating the flight control analysis templates with the default airframes, consider customizing the templates for your own airframe model.

Flight Control Analysis Templates

To familiarize yourself with Aerospace Blockset flight control analysis templates:

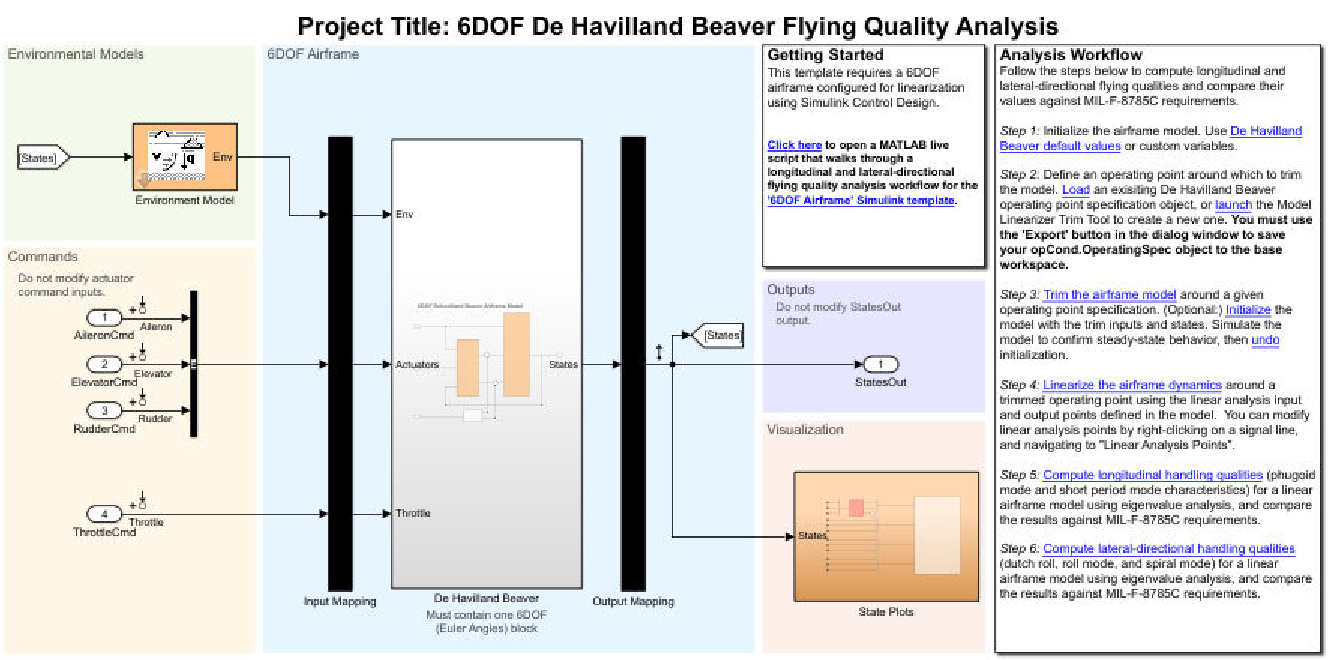
1 Open one of the templates. For example, to open a 3DOF template:

```
asbFlightControlAnalysis('3DOF')
```

To open a 6DOF template:

```
asbFlightControlAnalysis('6DOF')
```

The flight control analysis model opens.



2 The **Analysis Workflow** section contains a clickable guided workflow to compute longitudinal and lateral-directional flying qualities and compare their values against MIL-F-8785C requirements. Each step creates the necessary variables for this step. To perform the flying quality analysis, sequentially click the links in the steps.

- **a** Create an operating point specification object in the base workspace for the airframe model using the Model Linearizer. Alternatively, load the default object provided in step 2.
- **b** To trim the airframe, click **Trim the airframe** in step 3. This action calls the `trimAirframe` function.
- **c** To linearize the airframe around a trimmed operating point, click **Linearize the airframe** in step 4. This action calls the `linearizeAirframe` function.
- **d** To compute the longitudinal flying qualities, click **Compute longitudinal handling qualities**. This action calls the `computeLongitudinalFlyingQualities` function.
- **e** To compute the lateral-directional handling qualities, click **Compute lateral-directional handling qualities** in step 6. This action calls the `computeLateralDirectionalFlyingQualities` function.

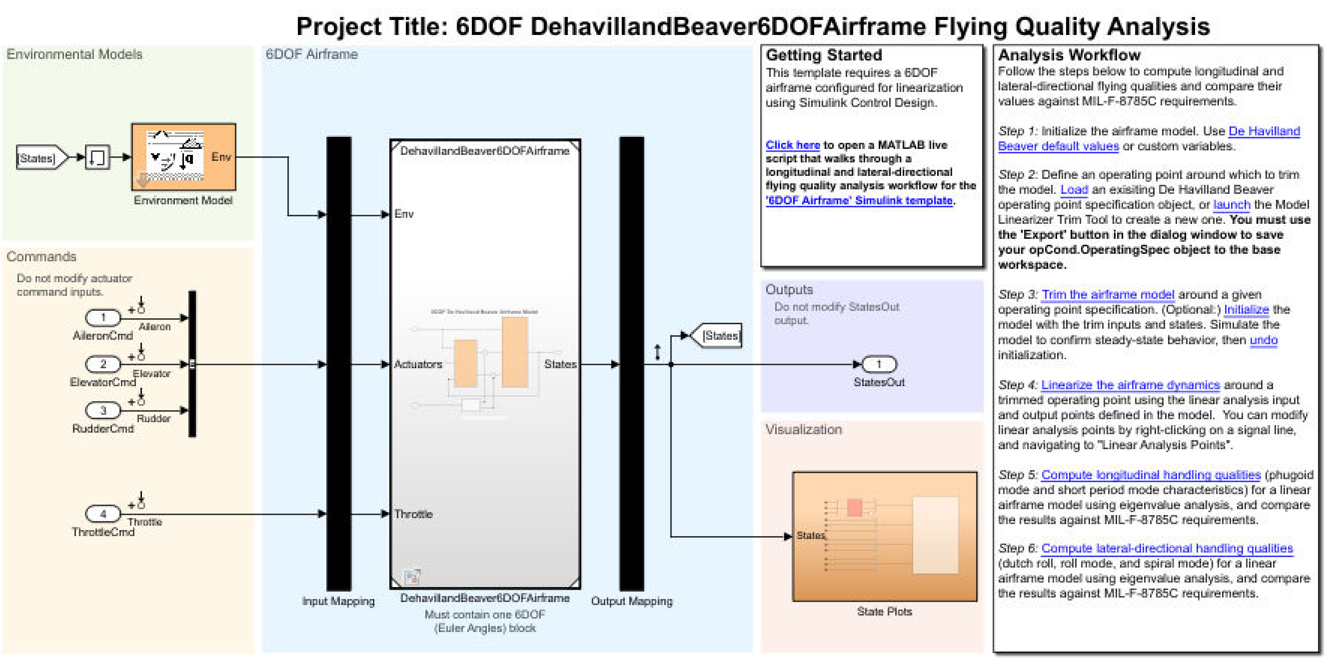
Modify Flight Control Analysis Templates

When you are comfortable using the 3DOF and 6DOF flight control analysis templates on page 2-66 to trim, linearize, and compute the longitudinal and lateral-directional handling qualities for the default airframes, consider customizing the templates to include your own airframe.

1. Open a 3DOF or 6DOF template and change the airframe to one of your own. For example, to change the template airframe to an external model:

   ```
   asbFlightControlAnalysis('6DOF', 'sixDOFAirframeExample','DehavillandBeaver6DOFAirframe')
   ```

   This command replaces the de Havilland Beaver subsystem with the `DehavillandBeaver6DOFAirframe` model and includes it as a referenced model.



   Alternatively, in the corresponding canvas, manually replace the default model airframe in the blue area with your own airframe.

2. On the canvas, align the inputs and outputs of the airframe using the Input Mapping and Output Mapping subsystems.

3. Create a new operating point specification object. In the **Analysis Workflow** section, go to step 2 and click **Launch** to start the Model Linearizer.
4. To save your `opCond.OperatingSpec` object to the base workspace, click **Export** in the dialog window.
5. To trim, linearize, and compute the longitudinal and lateral-directional handling qualities for the airframe model, click the links in workflow steps 3, 4, 5, and 6.

Explore Flight Control Analysis Functions

The flight control analysis live scripts and template workflows use these functions:

- `asbFlightControlAnalysis`
- `trimAirframe`
- `linearizeAirframe`
- `computeLongitudinalFlyingQualities`
- `computeLateralDirectionalFlyingQualities`

To customize your own scripts to trim airframes around operating points, linearize airframes, and calculate longitudinal and lateral-directional handling qualities, you can use these functions in a workflow:

1. Create a flight control analysis template using the `asbFlightControlAnalysis` function.
2. Trim the airframe model around an operating point using the `trimAirframe` function.

   This step creates a trimmed operating point, which the `linearizeAirframe` function requires.
3. Linearize the airframe model around the trimmed operating point using the `linearizeAirframe` function.

   This step creates a state space model that describes the linearized dynamics of the airframe model at a trimmed operating point.
4. Compute the flying qualities for the airframe, including short- and long-period (phugoid) mode characteristics of the specified state space model, using `computeLongitudinalFlyingQualities`. Compute lateral-directional (Dutch roll, roll, and spiral) mode characteristics, using `computeLateralDirectionalFlyingQualities`.

For example:

```
asbFlightControlAnalysis('6DOF', 'DehavillandBeaverAnalysisModel');
opSpecDefault = DehavillandBeaver6DOFOpSpec('DehavillandBeaverAnalysisModel');
opTrim = trimAirframe('DehavillandBeaverAnalysisModel', opSpecDefault);
linSys = linearizeAirframe('DehavillandBeaverAnalysisModel', opTrim);
lonFlyingQual = computeLongitudinalFlyingQualities('DehavillandBeaverAnalysisModel', linSys)
latFlyingQual = computeLateralDirectionalFlyingQualities('DehavillandBeaverAnalysisModel', linSys)
```

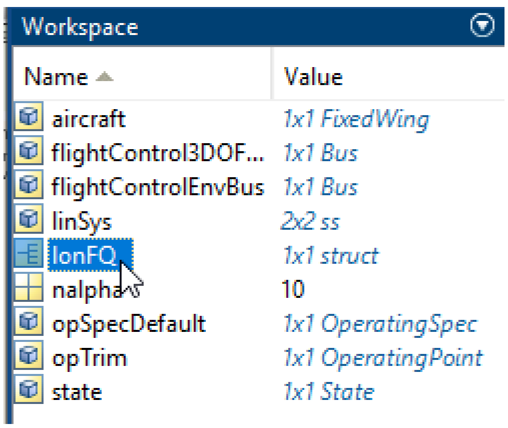
Plot Short-Period Undamped Natural Frequency Results

After computing the lateral-directional handling qualities, you can plot the short-period undamped natural frequency response ω_{nSP} using the `shortPeriodCategoryAPlot` function. To plot the category B or category C flight phase, use the `shortPeriodCategoryBPlot` or `shortPeriodCategoryCPlot` function. This example describes how to plot the short-period undamped natural frequency response for the Sky Hogg model.

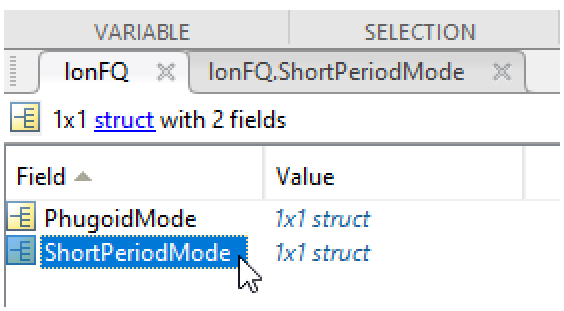
1. Start the flight control analysis template for the 3DOF configuration.

   ```
   asbFlightControlAnalysis('3DOF')
   ```

   The `3DOF Sky Hogg Longitudinal Flying Quality Analysis` project starts in the Simulink Editor.
2. To compute longitudinal and lateral-directional flying qualities, in the **Analysis Workflow** section, click through the guided workflow, click **OK** when prompted.
3. After computing longitudinal and lateral-directional flying qualities, find and double-click the `lonFQ` structure in your workspace.

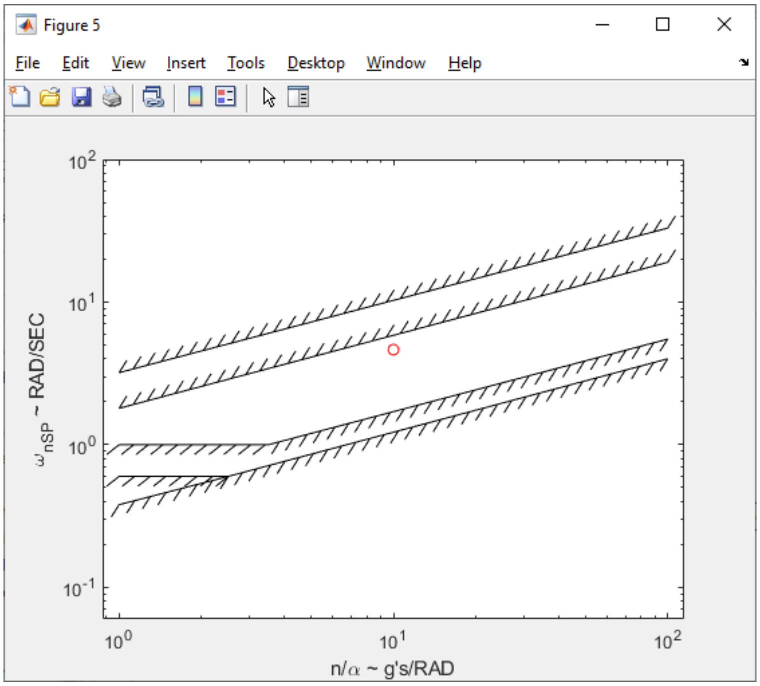
   In the variables viewer, double-click the `ShortPeriodMode` variable.

4. Check that the `wn` variable exists. The `wn` variable is the short-period undamped natural frequency response you want to plot.
5. Plot the short-period undamped natural frequency response. In the MATLAB Command Window, use the `shortPeriodCategoryAPlot` function. For example, for a load factor per angle of attack of `10`, enter this command.

   ```
   shortPeriodCategoryAPlot(10, lonFQ.ShortPeriodMode.wn, 'ro')
   ```

   A figure window with the plotted short-period undamped natural frequency response displays.



6 To evaluate if the results are within your tolerance limits, check that the red dot is within your limits.

See Also

`asbFlightControlAnalysis` | `computeLateralDirectionalFlyingQualities` | `computeLongitudinalFlyingQualities` | `linearizeAirframe` | `trimAirframe` | `shortPeriodCategoryAPlot` | `shortPeriodCategoryBPlot` | `shortPeriodCategoryCPlot` | Model Linearizer

Model Spacecraft

To model, simulate, and visualize the motion and dynamics of spacecraft, use the Spacecraft library blocks. Use the Spacecraft Dynamics sublibrary blocks to define spacecraft constellation dynamics, orbit propagation, or attitude profile. Use the CubeSat Vehicles sublibrary blocks to define a single CubeSat vehicle.

The Spacecraft Dynamics library contains:

- Spacecraft Dynamics block — Models translational and rotational dynamics of spacecraft using numerical integration. It computes the position, velocity, attitude, and angular velocity of one or more spacecraft over time.

  You can define orbital states as a set of orbital elements or as position and velocity state vectors in an inertial (ICRF) or fixed-frame (FF) coordinate system. To propagate orbital states, the block uses the gravity model defined in the "Central Body" section. It also uses external accelerations and forces provided as inputs to the block.

  You can define attitude states using quaternions, direction cosine matrices (DCMs), or Euler angles. To propagate attitude states, the block uses moments provided as inputs to the block and mass properties defined in the "Mass" section.
- Orbit Propagator — Propagates the orbit of one or more spacecraft by a propagation method. The library contains two versions of the Orbit Propagator block, each preconfigured for a different propagation method, Kepler (unperturbed) or Numerical (high precision). The Kepler (unperturbed) version of the block uses a universal variable formulation propagation method that is considered faster. The Numerical (high precision) version of the block uses a numerical integration propagation method, which is considered more accurate and therefore, slower.

  To use, define orbital states as a set of orbital elements or as position and velocity state vectors in an inertial (ICRF) or fixed-frame coordinate system.
- Attitude Profile — Calculate the shortest quaternion rotation that aligns the primary alignment vector with the primary constraint vector.

  Provide the primary constraint as either a pointing mode, or via a custom constraint vector. The block then aligns secondary alignment and constraint vectors as much as possible without breaking primary alignment.

The CubeSat Vehicles library contains the CubeSat Vehicle, which provides a high level mission planning/rapid prototyping option to quickly model and propagate satellite orbits, one satellite at a time. For more information on the CubeSat Vehicle block, see “Model and Simulate CubeSats” on page 2-73. To propagate multiple satellites simultaneously, see the Orbit Propagator block.

See Also

Attitude Profile | CubeSat Vehicle | Orbit Propagator | Spacecraft Dynamics

Related Examples

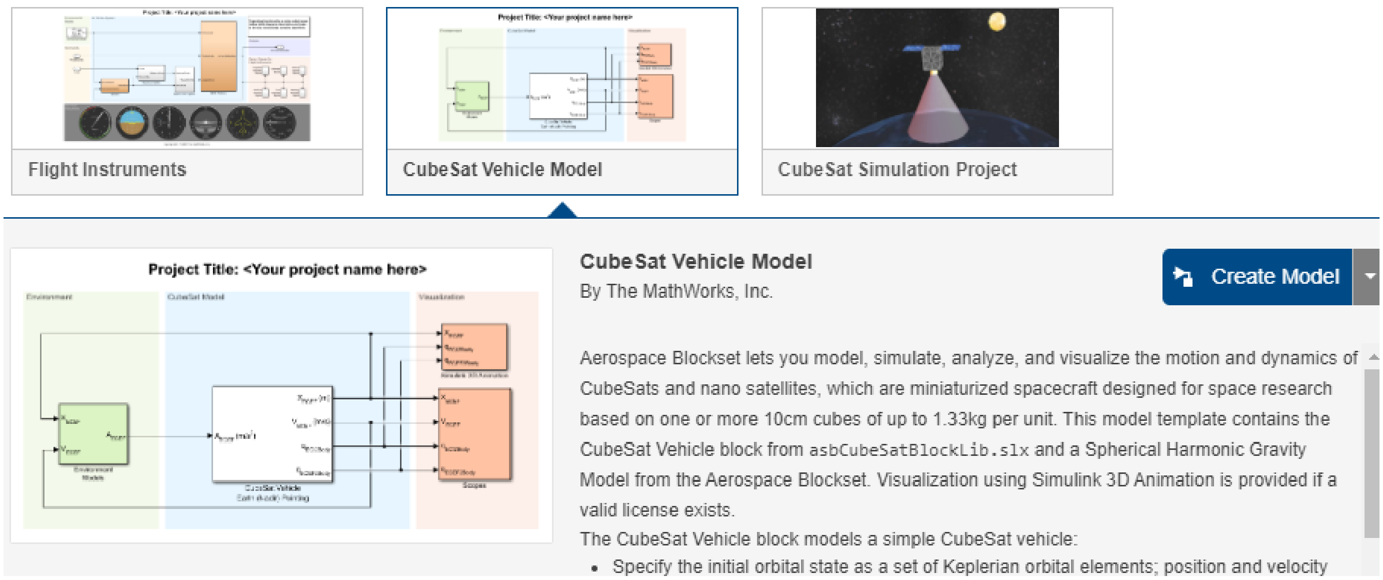
- “Model and Simulate CubeSats” on page 2-73
- “Getting Started with the Spacecraft Dynamics Block” on page 9-119
- “Constellation Modeling with the Orbit Propagator Block” on page 9-96

- “Developing the Apollo Lunar Module Digital Autopilot” on page 9-147

Model and Simulate CubeSats

To create models, use the CubeSat Vehicle blocks, model template, and project. Explore the spacecraft example modeling multiple spacecraft. The CubeSat Vehicle block propagates one satellite at a time. To propagate multiple satellites simultaneously, use the Orbit Propagator block. To calculate shortest quaternion rotation, use the Attitude Profile block.

To help you get started modeling and simulating spacecraft, Aerospace Blockset provides a project and model on the Simulink Start Page.



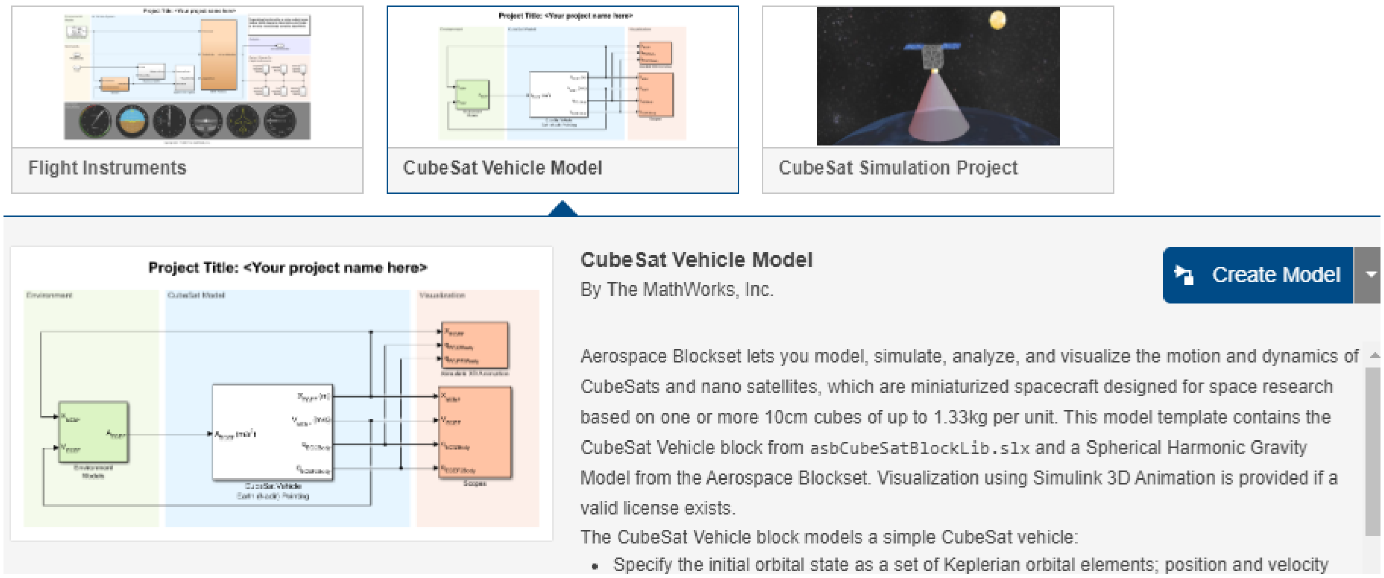
- CubeSat Vehicle Model template — A model template (`CubeSat Simulation Project`) that illustrates how to propagate and visualize CubeSat trajectories using the CubeSat Vehicle block. The Spherical Harmonic Gravity Model block is used as the gravitational potential source for orbit propagation. The preconfigured pointing modes set in the CubeSat Vehicle block control the attitude.
- CubeSat Simulation Project — A ready-to-simulate project (`CubeSat Simulation Project`) that illustrates how to create a detailed CubeSat system design in Simulink by adding in detailed vehicle components to the provided framework.
- CubeSat Model-Based System Engineering Project — A ready-to-simulate project (`CubeSat Model-based System Engineering Project`) that shows how to model a space mission architecture in Simulink with System Composer™ and Aerospace Blockset for a 1U CubeSat in low Earth orbit (LEO).

CubeSat Vehicle Model Template

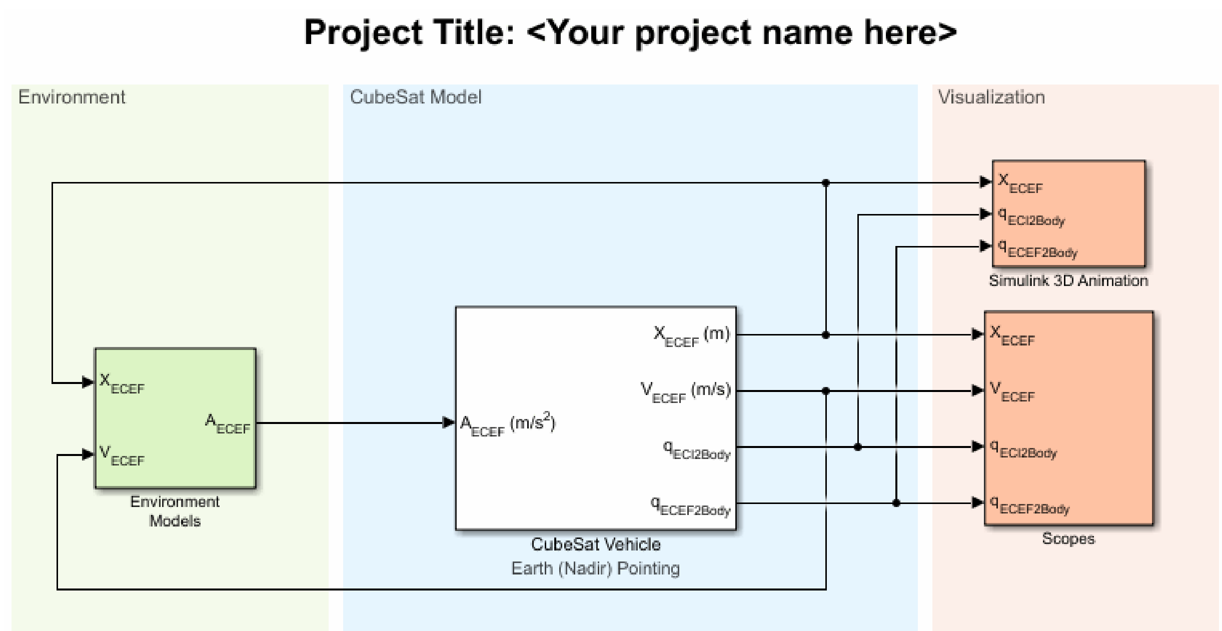
The template model is a ready-to-simulate example that contains a CubeSat Vehicle block with visualization using Simulink 3D Animation.

For a project that illustrates the use of the Vehicle Model block in place of the CubeSat Vehicle Model block, see “CubeSat Simulation Project” on page 2-75.

1 Start the CubeSat Vehicle Model template.



2 Click **Create Model**.



3 The CubeSat Vehicle block models a simple CubeSat vehicle that you can use as is, with the CubeSat Vehicle block configured to use the initial orbital state as a set of Keplerian orbital elements.

The model uses the Spherical Harmonic Gravity Model block to provide the vehicle gravity for the CubeSat.

To familiarize yourself with CubeSats, experiment with the CubeSat Vehicle block settings.

- On the **CubeSat Orbit** tab of the block, you can optionally use the **Input method** parameter to change the initial orbital state as a set of:

    - Position and velocity state vectors in Earth-centered inertial axes
    - Position and velocity state vectors in Earth-centered Earth-fixed axes
    - Latitude, longitude, altitude, and velocity of the body with respect to ECEF, expressed in the NED frame
  - On the **CubeSat Attitude** tab, you can specify the alignment and constraint vectors to define the CubeSat attitude control.
    - The CubeSat vehicle first aligns the primary alignment vector with the primary constraint vector. The CubeSat vehicle then attempts to align the secondary alignment vector with the secondary constraint vector as closely as possible without affecting primary alignment.
    - The CubeSat Altitude tab also lets you choose between preconfigured Earth (Nadir) Earth Pointing (default) and Sun Tracking attitude control modes.
  - On the **Earth Orientation Parameters** tab, you can direct the block to include higher order earth orientation parameters in transformations between the ECI and ECEF coordinate systems.
4. Run and simulate the model.
5. To view the output signals from the CubeSat, double-click the Scopes subsystem and open the multiple scopes.
6. If you have a valid Simulink 3D Animation license, you can also visualize the orbit in the **CubeSat Orbit Animation** window.
7. Save a copy of the orbit propagation model. You can use this model for the mission analysis live script.

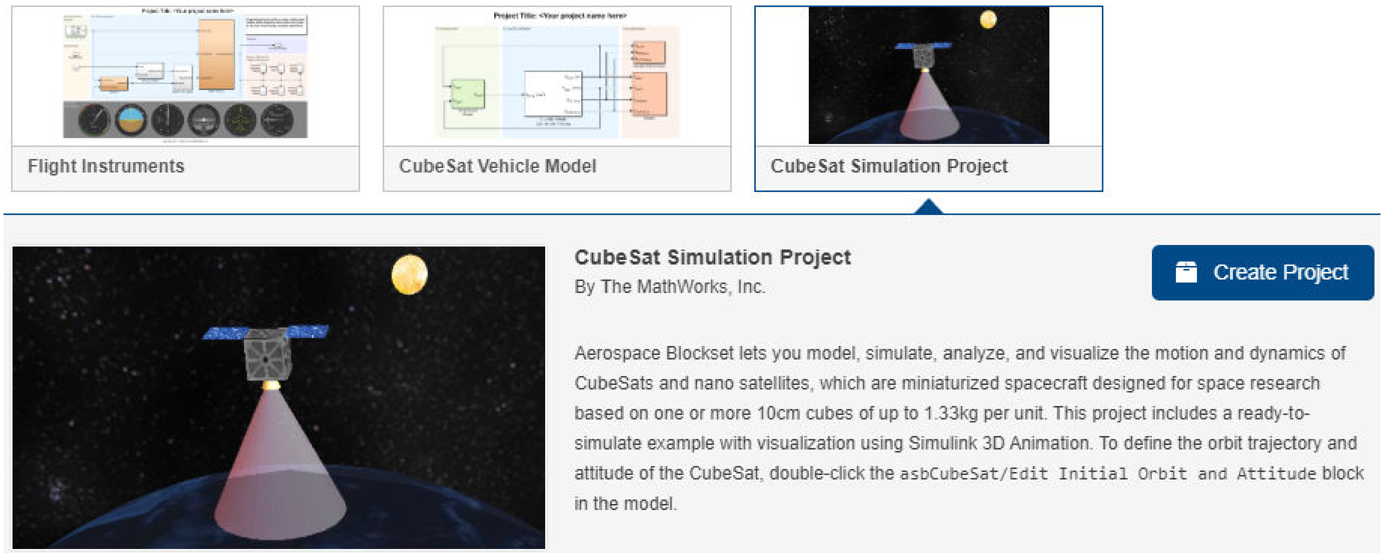
The CubeSat Vehicle Model template CubeSat Vehicle block uses simple preconfigured orbit and attitude control modes. To model and simulate CubeSat vehicles using your own detailed components, consider the CubeSat Simulation Project from the Simulink Start Page. For more information, see “CubeSat Simulation Project” on page 2-75

CubeSat Simulation Project

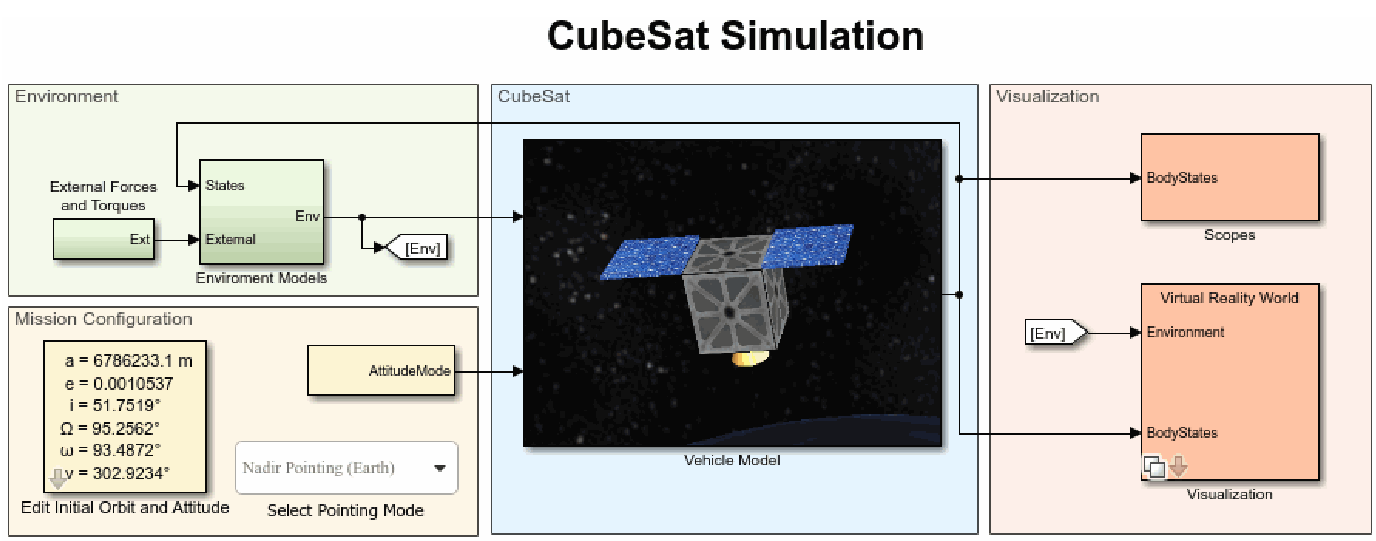
The project is a ready-to-simulate example with visualization using Simulink 3D Animation. This example uses a Vehicle Model subsystem in place of a CubeSat Vehicle block.

For a model that also models a space mission architecture with System Composer, see “CubeSat Model-Based System Engineering Project” on page 2-78.

1. Start the CubeSat Simulation Project.



2 Click **Create Project** and follow the instructions.



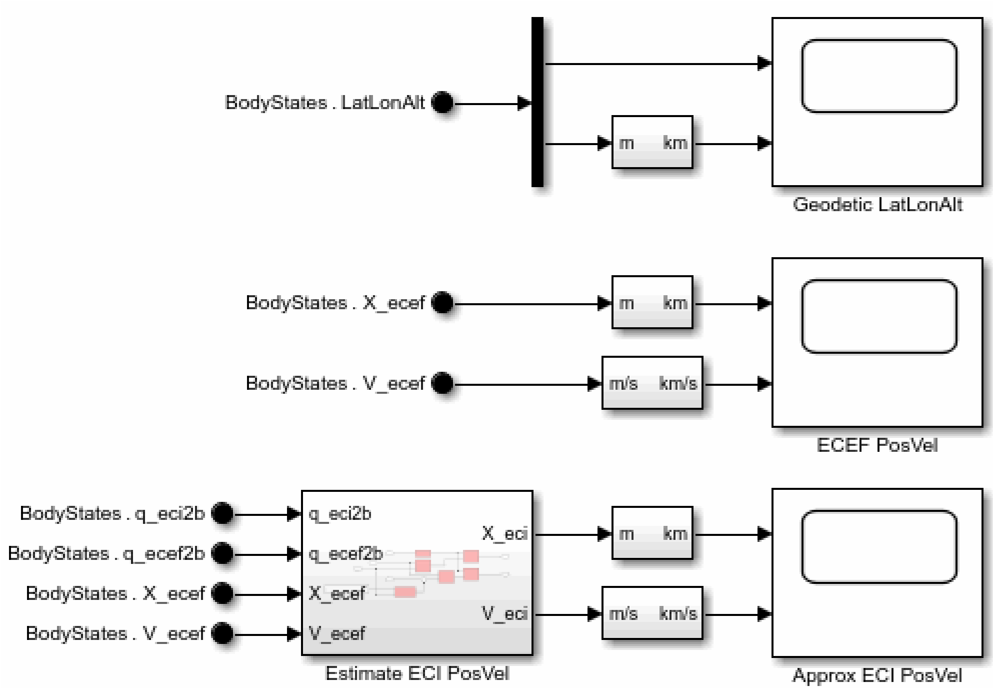
3 The Vehicle Model subsystem models a CubeSat vehicle that you can use as is.

To create your own more sophisticated satellite models, experiment with the Vehicle Model framework. For example, you can replace the perfect thruster model included by default in the actuator subsystem with your own more realistic thruster or reaction wheel model.

4 To change the orbit trajectory and attitude of the CubeSat, in the Mission Configuration section, double-click the Edit Initial Orbit and Attitude block. These parameters have the same intent as the corresponding parameters as the CubeSat Vehicle block.

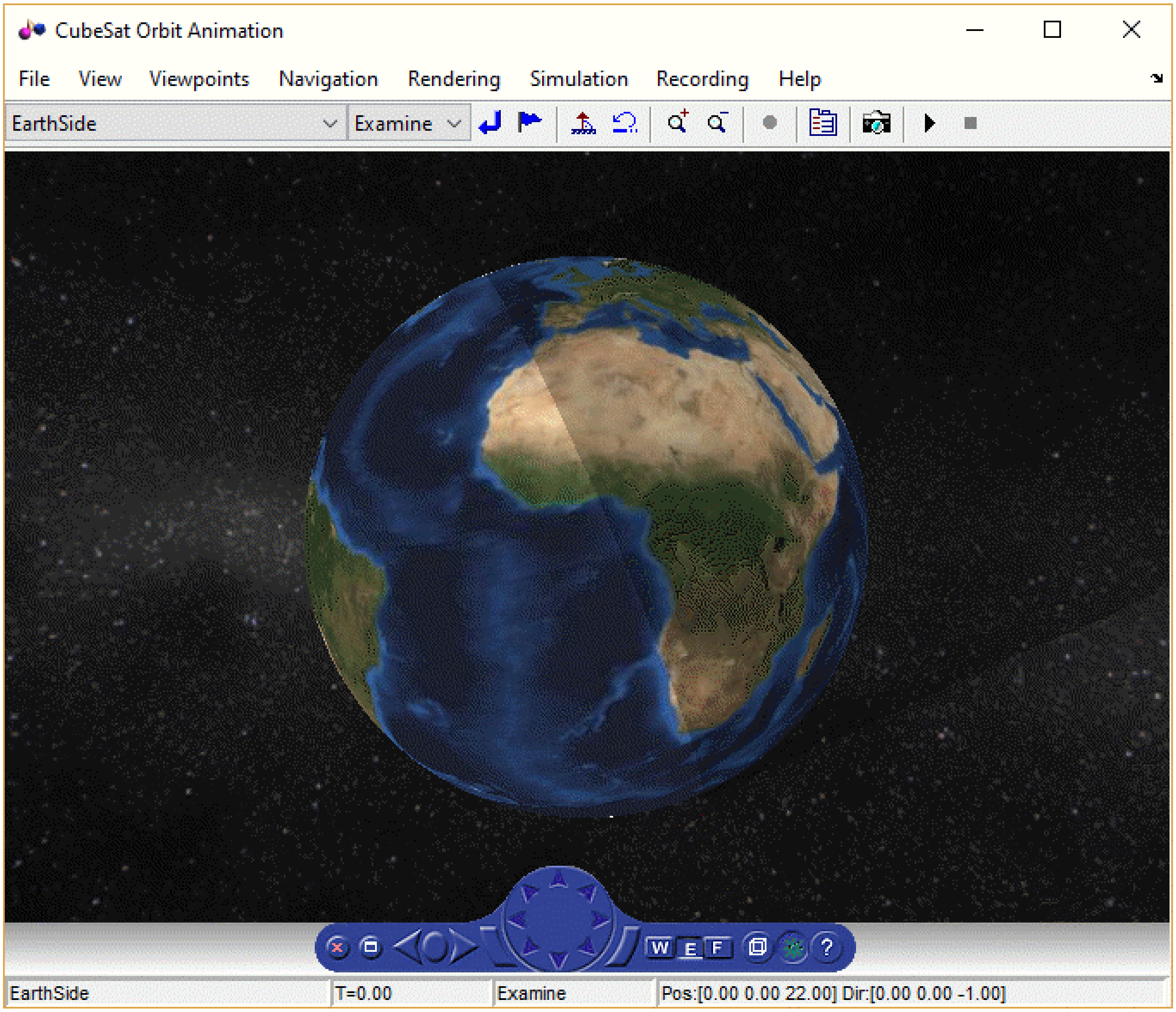
5 Run and simulate the model.

6 To view the output signals from the CubeSat, double-click the Scopes subsystem and open the multiple scopes.



7 If you have a license for Simulink 3D Animation, you can also visualize the orbit in an animation window. Double-click the Visualization subsystem and click the **Open Simulink 3D Animation window** button.

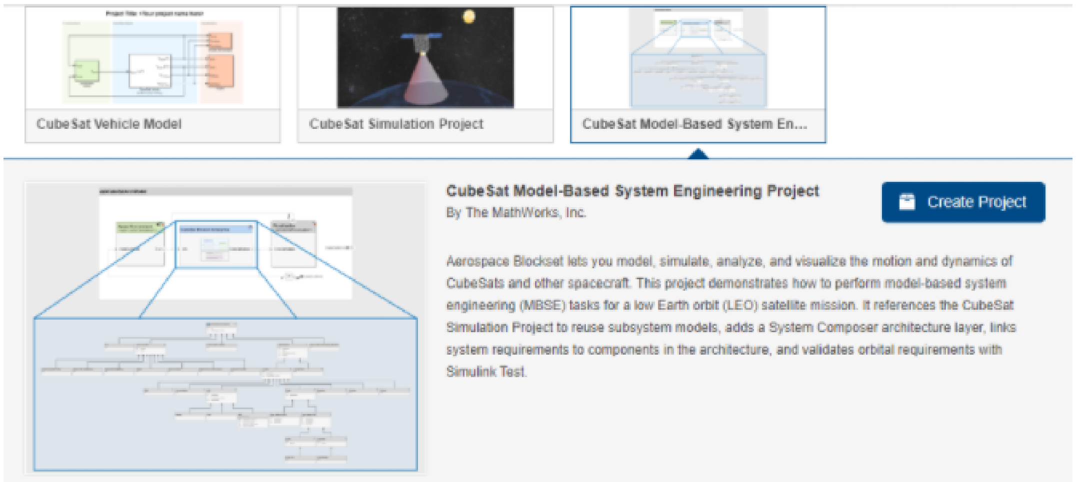
The **CubeSat Orbit Animation** window opens.



CubeSat Model-Based System Engineering Project

The CubeSat Model-Based System Engineering (MBSE) Project is a simulation-ready example that shows how to model a space mission architecture with System Composer and Aerospace Blockset. The project references the "CubeSat Simulation Project" on page 2-75 to reuse subsystem models, then adds a System Composer architecture layer, links system requirements to components in the architecture, and verifies the top-level mission requirement with Simulink Test™. The project visualizes results using Simulink 3D Animation, Aerospace Toolbox satellite scenarios, and Mapping Toolbox™.

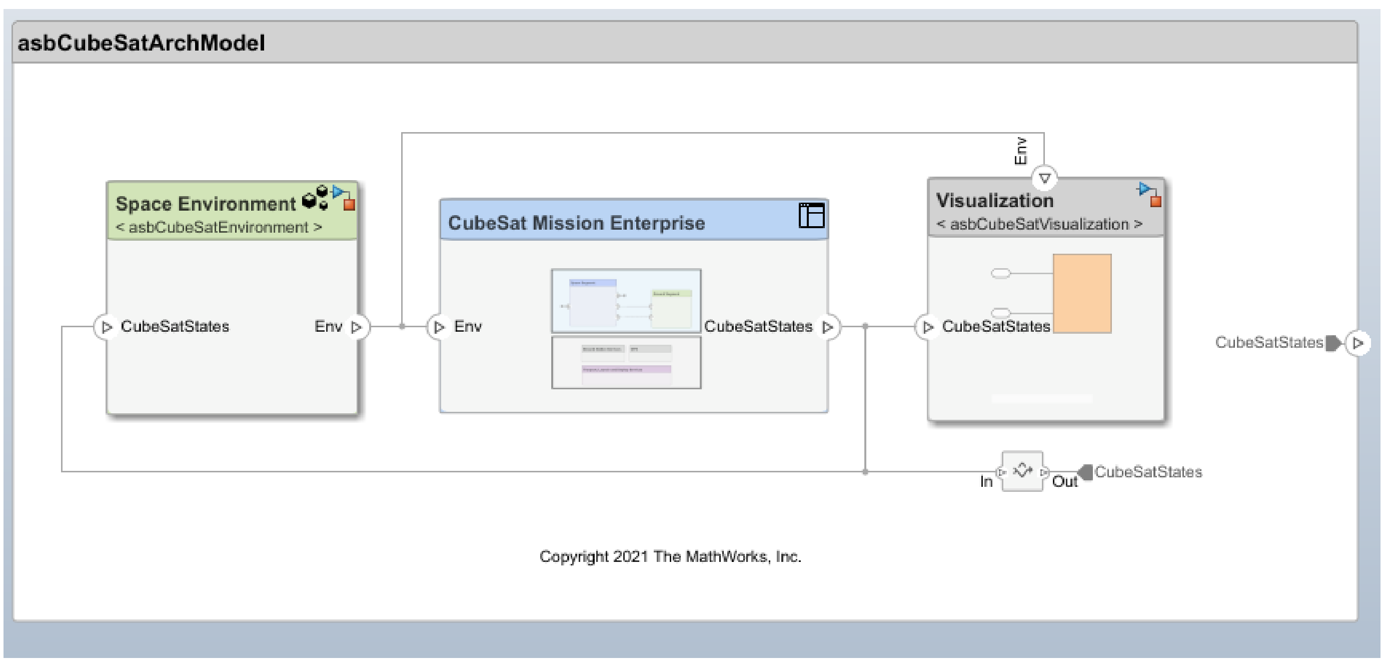
1 Open the CubeSat Model-Based System Engineering Project, click **Create Project**, and follow the instructions.



2 From the **Projects Shortcut** tab in the MATLAB Command Window, click **MBSE Template Overview**.

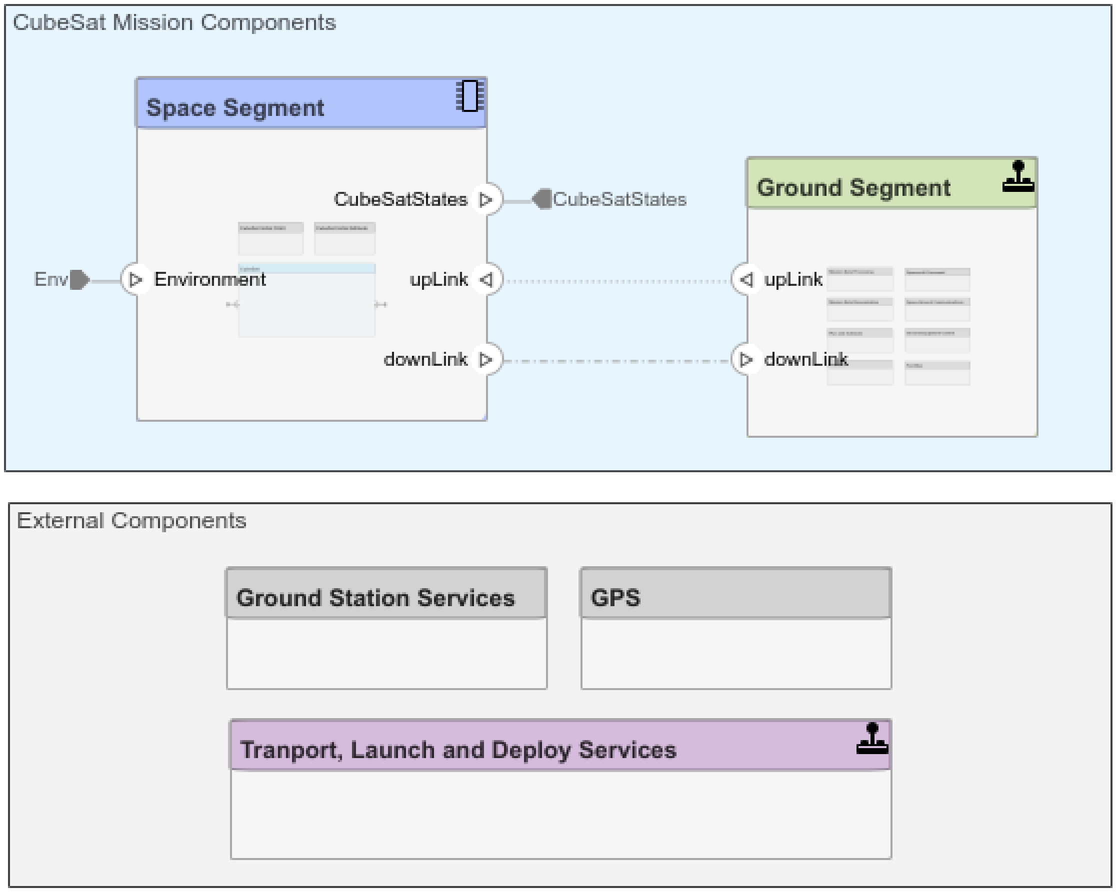
   The template overview describes the project and how to model a space mission architecture.

3 Use the overview to explore the `asbCubeSatArchModel` architecture and learn how to extend the architecture using System Composer.



   The project helps define an architecture within System Composer. The architecture in this example is based on CubeSat Reference Model (CRM) developed by the International Council on Systems Engineering (INCOSE) Space Systems Working Group (SSWG) (https://www.incose.org/communities/working-groups-initiatives/space-systems).

4 To view the underlying parts of a component, double-click the component. For example, to view the architecture model for the mission, double-click `CubeSat Mission Enterprise`.

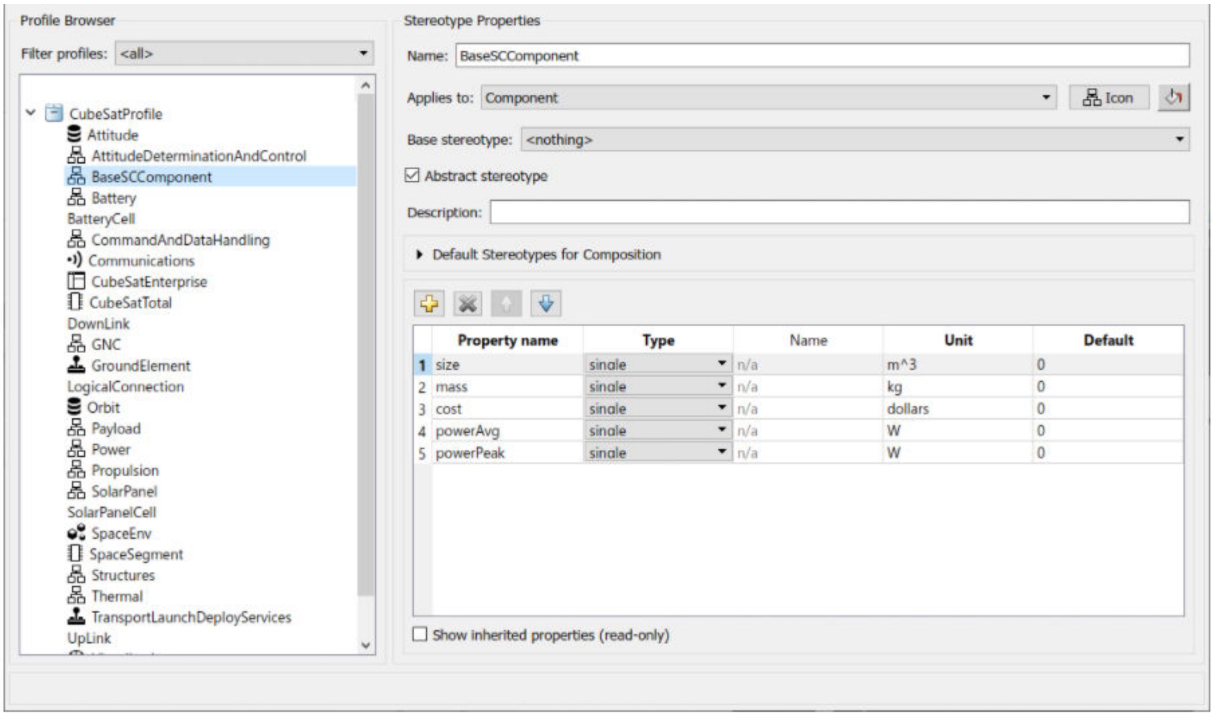


This model consists of System Composer components that model the mission enterprise.

5 Use the **MBSE Template Overview** to navigate the project and learn how to use System Composer elements to model the space mission architecture.

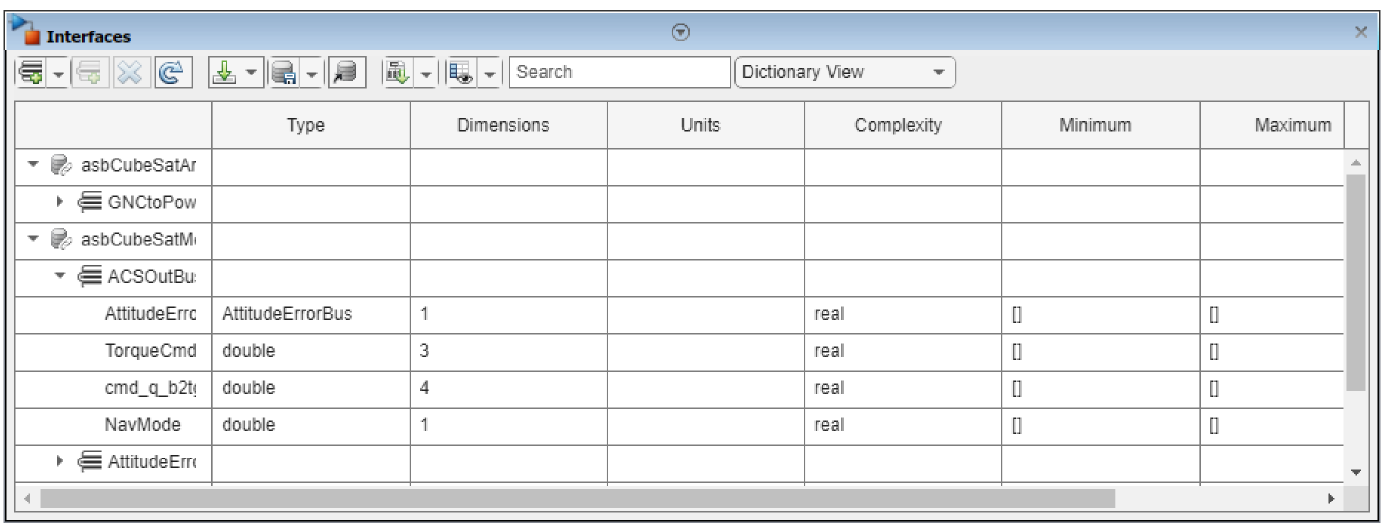
With System Composer:

a Extend architecture elements by adding domain-specific metadata to the element using stereotypes. Apply stereotypes to components, connectors, ports, and other architecture elements to provide these elements with a common set of properties. To view, edit, or add new stereotypes to a profile, on the **Modeling** tab, click **Profile Editor** and select a stereotype profile. For this example, open the `CubeSatProfile.xml` profile.

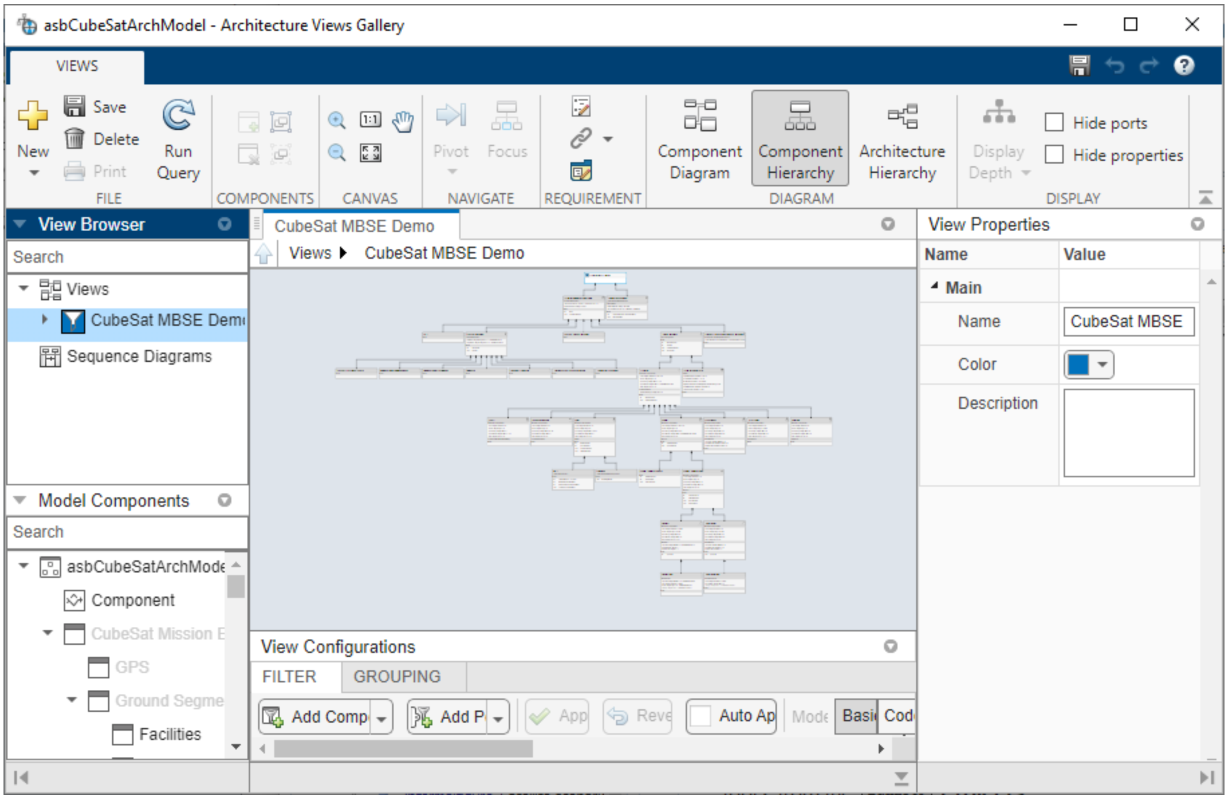


You can modify a profile, add new profiles, and apply new profiles to a component.

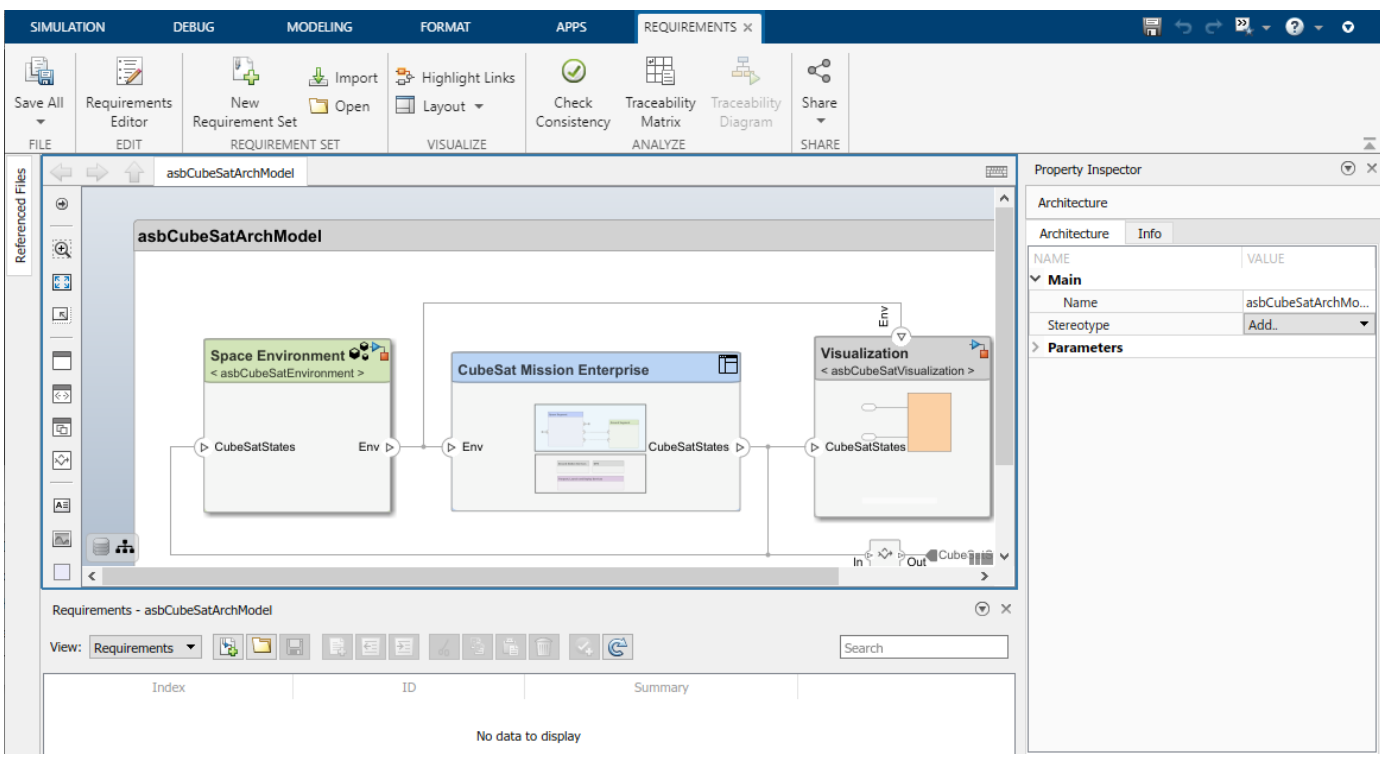
b Define the kind of information that flows through a port using interfaces. To view, edit, or add new interfaces to a port, on the **Modeling** tab, click **Interface Editor** and select an interface. For example, select **asbCubeSatModelData.sldd** > **ACSOutBus**.



c Visualize the system with an Architecture view by clicking **Views** > **Architecture Views** and selecting a view, for example, **CubeSat MBSE Demo**.

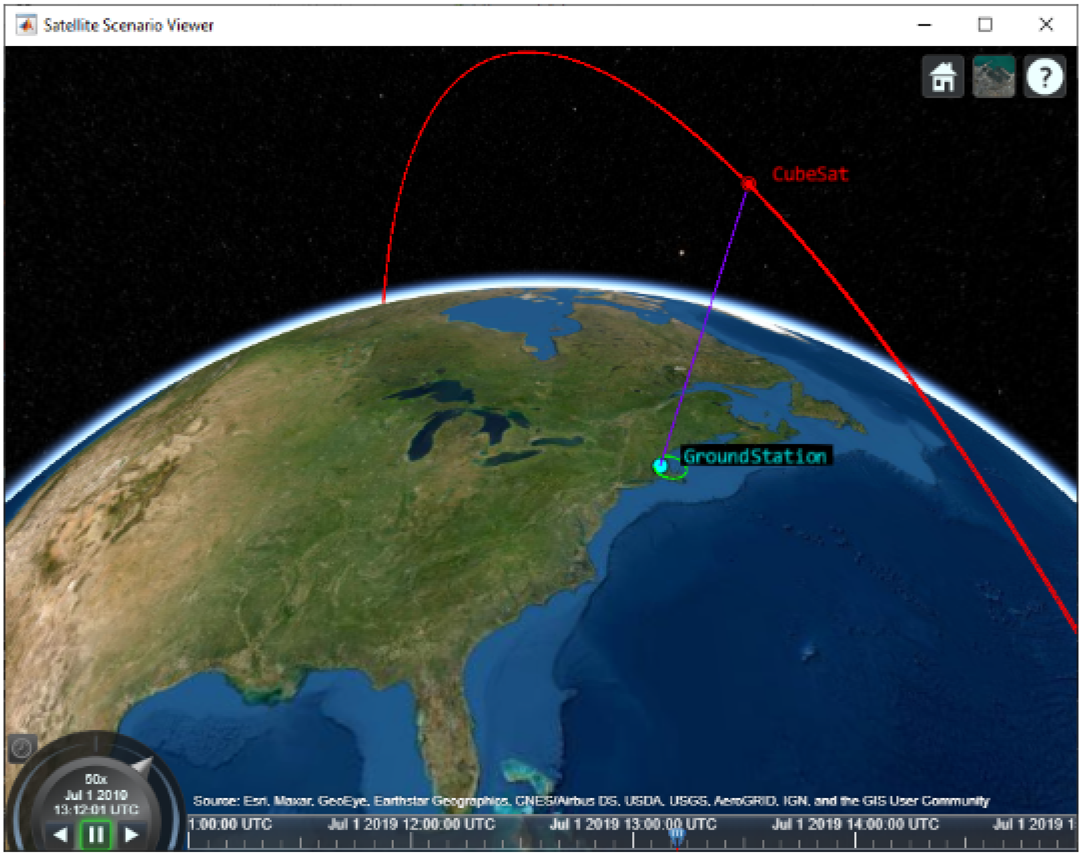


d To establish traceability, link system requirements by allocating functional requirements to components. In the **Apps** tab of the model, select **Requirements Manager**.



   e. For more information on working with this project, such as connecting the architecture to design models or simulating the architecture, see the **MBSE Template Overview**.

6. To validate the top-level mission requirement, use Simulink Test.
7. To understand how to perform a mission analysis of the CubeSat using the satellite scenario tools, from the **Projects Shortcut** tab in the MATLAB Command Window, click **Analyze with Satellite Scenario**.



Utility Functions

Aerospace Toolbox provides utility functions for coordinate transformations. You can use these functions to go between the various initial condition modes of the CubeSat Vehicle block.

Action	Function
Calculate position and velocity vectors in Earth-centered inertial mean-equator mean-equinox	`ecef2eci`
Calculate position, velocity, and acceleration vectors in Earth-centered Earth-fixed (ECEF) coordinate system	`eci2ecef`
Calculate Greenwich mean and apparent sidereal times	`siderealTime`

Action	Function
Calculate Keplerian orbit elements using geocentric equatorial position and velocity vectors	`ijk2keplerian`
Calculate position and velocity vectors in geocentric equatorial coordinate system using Keplerian orbit elements	`keplerian2ijk`

References

[1] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

See Also

Attitude Profile | CubeSat Vehicle | Orbit Propagator | `ecef2eci` | `eci2ecef` | `ijk2keplerian` | `keplerian2ijk` | `siderealTime`

More About

- "Model-Based Systems Engineering for Space-Based Applications" on page 9-187

Case Studies

Ideal Airspeed Correction

In this section...

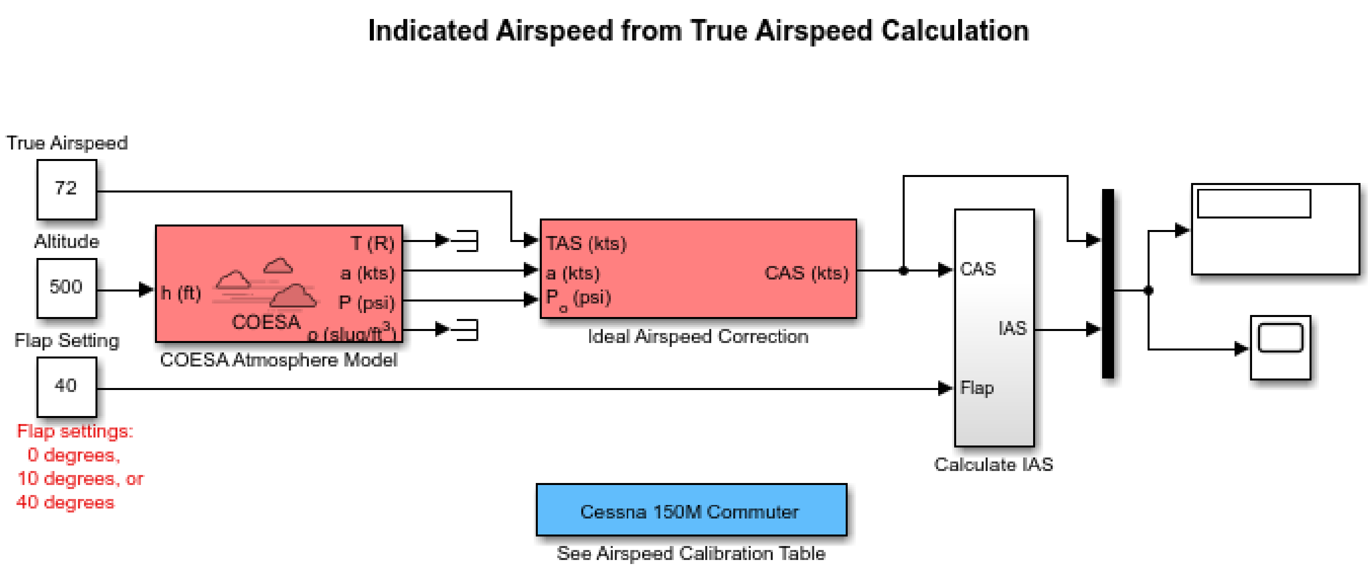
Introduction

This case study simulates indicated and true airspeed. It constitutes a fragment of a complete aerodynamics problem, including only measurement and calibration.

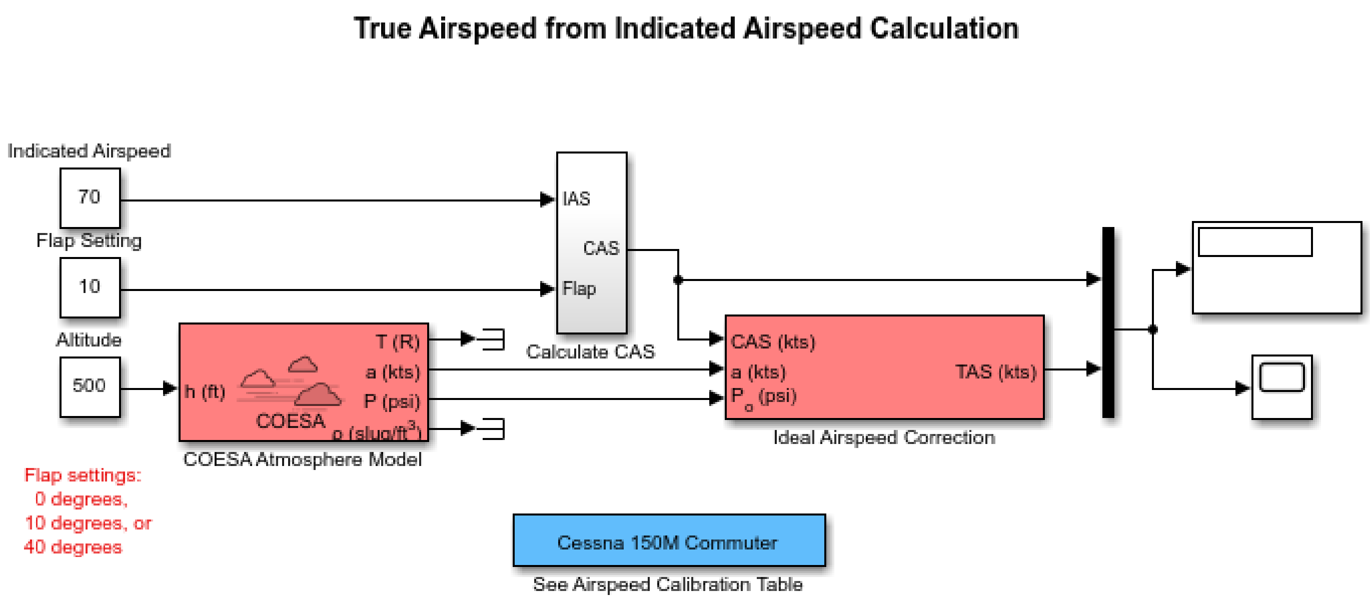
Airspeed Correction Models

To view the airspeed correction models, enter the following at the MATLAB command line:

```
openExample('aeroblk_indicated')
aeroblk_calibrated
```



aeroblk_indicated Model



aeroblk_calibrated Model

Measure Airspeed

To measure airspeed, most light aircraft designs implement pitot-static airspeed indicators based on Bernoulli's principle. Pitot-static airspeed indicators measure airspeed by an expandable capsule that expands and contracts with increasing and decreasing dynamic pressure. This is known as *calibrated airspeed* (CAS). It is what a pilot sees in the cockpit of an aircraft.

To compensate for measurement errors, it helps to distinguish three types of airspeed. These types are explained more completely in the following.

Airspeed Type	Description
Calibrated	Indicated airspeed corrected for calibration error
Equivalent	Calibrated airspeed corrected for compressibility error
True	Equivalent airspeed corrected for density error

Calibration Error

An airspeed sensor features a static vent to maintain its internal pressure equal to atmospheric pressure. Position and placement of the static vent with respect to the angle of attack and velocity of the aircraft determines the pressure inside the airspeed sensor and therefore the calibration error. Thus, a calibration error is specific to an aircraft's design.

An airspeed calibration table, which is usually included in the pilot operating handbook or other aircraft documentation, helps pilots convert the indicated airspeed to the calibrated airspeed.

Compressibility Error

The density of air is not constant, and the compressibility of air increases with altitude and airspeed, or when contained in a restricted volume. A pitot-static airspeed sensor contains a restricted volume of air. At high altitudes and high airspeeds, calibrated airspeed is always higher than equivalent airspeed. Equivalent airspeed can be derived by adjusting the calibrated airspeed for compressibility error.

Density Error

At high altitudes, airspeed indicators read lower than true airspeed because the air density is lower. True airspeed represents the compensation of equivalent airspeed for the density error, the difference in air density at altitude from the air density at sea level, in a standard atmosphere.

Model Airspeed Correction

The `aeroblk_indicated` and `aeroblk_calibrated` models show how to take true airspeed and correct it to indicated airspeed for instrument display in a Cessna 150M Commuter light aircraft. The `aeroblk_indicated` model implements a conversion to indicated airspeed. The `aeroblk_calibrated` model implements a conversion to true airspeed.

Each model consists of two main components:

- "COESA Atmosphere Model Block" on page 3-4 calculates the change in atmospheric conditions with changing altitude.
- "Ideal Airspeed Correction Block" on page 3-4 transforms true airspeed to calibrated airspeed and vice versa.

COESA Atmosphere Model Block

The COESA Atmosphere Model block is a mathematical representation of the U.S. 1976 COESA (Committee on Extension to the Standard Atmosphere) standard lower atmospheric values for absolute temperature, pressure, density, and speed of sound for input geopotential altitude. Below 32,000 meters (104,987 feet), the U.S. Standard Atmosphere is identical with the Standard Atmosphere of the ICAO (International Civil Aviation Organization).

The `aeroblk_indicated` and `aeroblk_calibrated` models use the COESA Atmosphere Model block to supply the speed of sound and air pressure inputs for the Ideal Airspeed Correction block in each model.

Ideal Airspeed Correction Block

The Ideal Airspeed Correction block compensates for airspeed measurement errors to convert airspeed from one type to another type. The following table contains the Ideal Airspeed Correction block's inputs and outputs.

Airspeed Input	Airspeed Output
True Airspeed	Equivalent airspeed
	Calibrated airspeed
Equivalent Airspeed	True airspeed
	Calibrated airspeed
Calibrated Airspeed	True airspeed
	Equivalent airspeed

In the `aeroblk_indicated` model, the Ideal Airspeed Correction block transforms true to calibrated airspeed. In the `aeroblk_calibrated` model, the Ideal Airspeed Correction block transforms calibrated to true airspeed.

The following sections explain how the Ideal Airspeed Correction block mathematically represents airspeed transformations:

- “True Airspeed Implementation” on page 3-5
- “Calibrated Airspeed Implementation” on page 3-5
- “Equivalent Airspeed Implementation” on page 3-5

True Airspeed Implementation

True airspeed (TAS) is implemented as an input and as a function of equivalent airspeed (EAS), expressible as

$$TAS = \frac{EAS \times a}{a_0\sqrt{\delta}}$$

where

α	Speed of sound at altitude in m/s
δ	Relative pressure ratio at altitude
a_0	Speed of sound at mean sea level in m/s

Calibrated Airspeed Implementation

Calibrated airspeed (CAS), derived using the compressible form of Bernoulli's equation and assuming isentropic conditions, can be expressed as

$$CAS = \sqrt{\frac{2\gamma P_0}{(\gamma-1)\rho_0}\left[\left(\frac{q}{P_0}+1\right)^{(\gamma-1)/\gamma}-1\right]}$$

where

ρ_0	Air density at mean sea level in kg/m^3
P_0	Static pressure at mean sea level in N/m^2
γ	Ratio of specific heats
q	Dynamic pressure at mean sea level in N/m^2

Equivalent Airspeed Implementation

Equivalent airspeed (EAS) is the same as CAS, except static pressure at sea level is replaced by static pressure at altitude.

$$EAS = \sqrt{\frac{2\gamma P}{(\gamma-1)\rho_0}\left[\left(\frac{q}{P}+1\right)^{(\gamma-1)/\gamma}-1\right]}$$

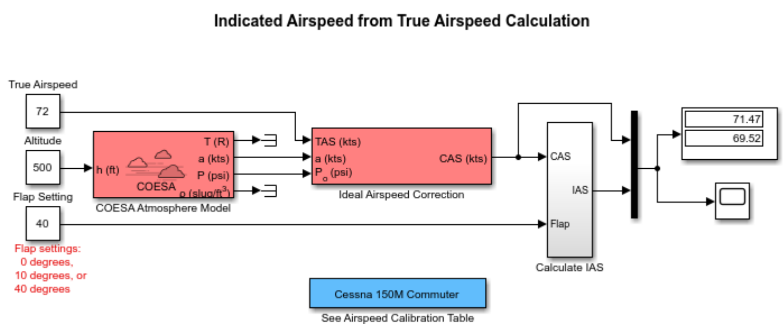
The symbols are defined as follows:

ρ_0	Air density at mean sea level in kg/m^3
P	Static pressure at altitude in N/m^2
γ	Ratio of specific heats
q	Dynamic pressure at mean sea level in N/m^2

Simulate Airspeed Correction

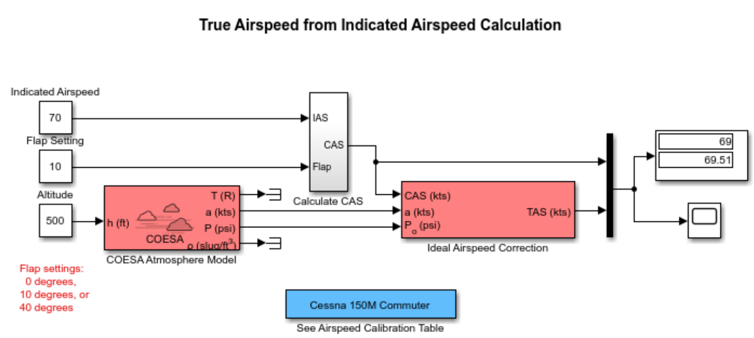
In the `aeroblk_indicated` model, the aircraft is defined to be traveling at a constant speed of 72 knots (true airspeed) and altitude of 500 feet. The flaps are set to 40 degrees. The COESA Atmosphere Model block takes the altitude as input and outputs the speed of sound and air pressure. Taking the speed of sound, air pressure, and airspeed as inputs, the Ideal Airspeed Correction block converts true airspeed to calibrated airspeed. Finally, the Calculate IAS subsystem uses the flap setting and calibrated airspeed to calculate indicated airspeed.

The model's Display block shows both indicated and calibrated airspeeds.



In the `aeroblk_calibrated` model, the aircraft is defined to be traveling at a constant speed of 70 knots (indicated airspeed) and altitude of 500 feet. The flaps are set to 10 degrees. The COESA Atmosphere Model block takes the altitude as input and outputs the speed of sound and air pressure. The `Calculate CAS` subsystem uses the flap setting and indicated airspeed to calculate the calibrated airspeed. Finally, using the speed of sound, air pressure, and true calibrated airspeed as inputs, the Ideal Airspeed Correction block converts calibrated airspeed back to true airspeed.

The model's Display block shows both calibrated and true airspeeds.



See Also

Related Examples

- "Indicated Airspeed from True Airspeed Calculation" on page 9-56
- "True Airspeed from Indicated Airspeed Calculation" on page 9-69

1903 Wright Flyer

In this section...

Introduction

Note The final section of this study requires the Simulink 3D Animation software.

This case study describes a model of the 1903 Wright Flyer. Built by Orville and Wilbur Wright, the Wright Flyer took to the skies in December 1903 and opened the age of controlled flight. The Wright brothers' flying machine achieved the following goals:

- Left the ground under its own power
- Moved forward and maintained its speed
- Landed at an elevation no lower than where it started

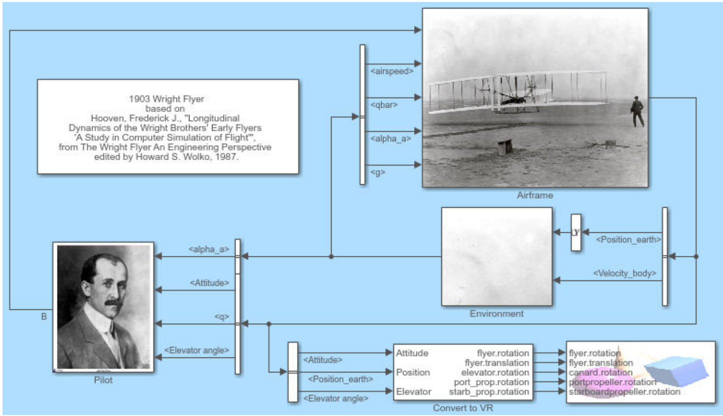
This model is based on an earlier simulation [1] that explored the longitudinal stability of the Wright Flyer and therefore modeled only forward and vertical motion along with the pitch angle. The Wright Flyer suffered from numerous engineering challenges, including dynamic and static instability. Laterally, the Flyer tended to overturn in crosswinds and gusts, and longitudinally, its pitch angle would undulate [2].

Under these constraints, the model recreates the longitudinal flight dynamics that pilots of the Wright Flyer would have experienced. Because they were able to control lateral motion, Orville and Wilbur Wright were able to maintain a relatively straight flight path.

Note, running this model generates information messages in the MATLAB Command Window and assertion warning messages in the Diagnostic Viewer. This is because the model illustrates the use of the Assertion block to indicate that the flyer is hitting the ground when landing.

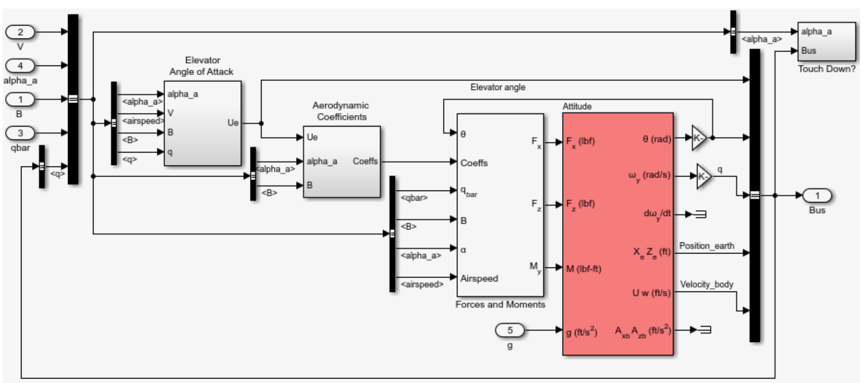
Wright Flyer Model

Open the Wright Flyer model by entering `openExample('aeroblk_wf_3dof')` at the MATLAB command line.



Airframe Subsystem

The Airframe subsystem simulates the rigid body dynamics of the Wright Flyer airframe, including elevator angle of attack, aerodynamic coefficients, forces and moments, and three-degrees-of-freedom equations of motion.

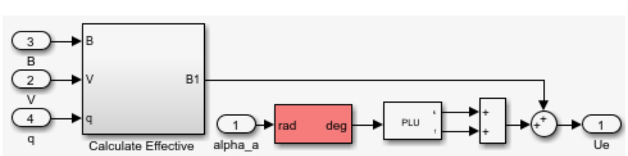


The Airframe subsystem consists of the following parts:

- “Elevator Angle of Attack Subsystem” on page 3-8
- “Aerodynamic Coefficients Subsystem” on page 3-9
- “Forces and Moments Subsystem” on page 3-9
- “3DOF (Body Axes) Block” on page 3-9

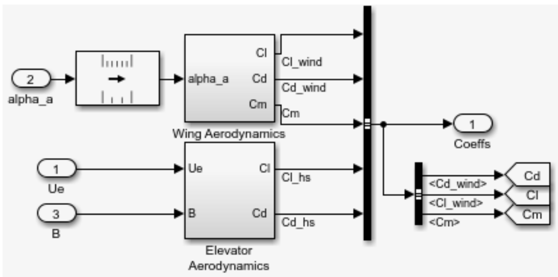
Elevator Angle of Attack Subsystem

The Elevator Angle of Attack subsystem calculates the effective elevator angle for the Wright Flyer airframe and feeds its output to the Pilot subsystem.



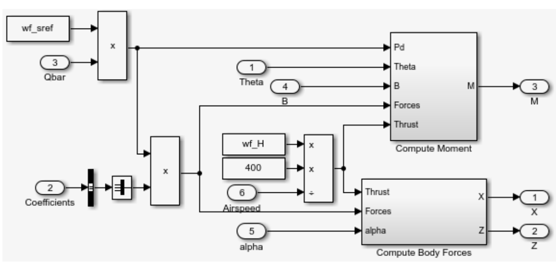
Aerodynamic Coefficients Subsystem

The Aerodynamic Coefficients subsystem contains aerodynamic data and equations for calculating the aerodynamic coefficients, which are summed and passed to the Forces and Moments subsystem. Stored in data sets, the aerodynamic coefficients are determined by interpolation using Prelookup blocks.



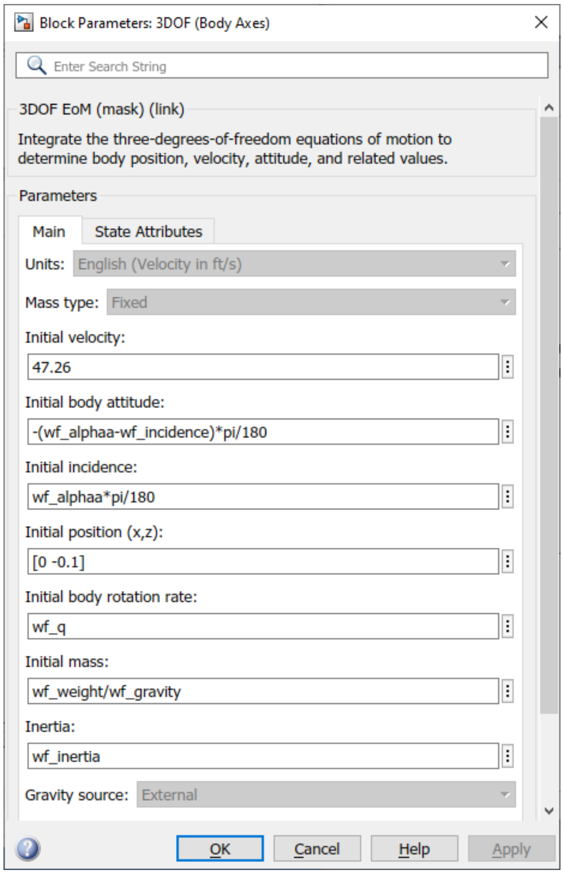
Forces and Moments Subsystem

The aerodynamic forces and moments acting on the airframe are generated from aerodynamic coefficients. The Forces and Moments subsystem calculates the body forces and body moments acting on the airframe about the center of gravity. These forces and moments depend on the aerodynamic coefficients, thrust, dynamic pressure, and reference airframe parameters.



3DOF (Body Axes) Block

The 3DOF (Body Axes) block use equations of motion to define the linear and angular motion of the Wright Flyer airframe. It also performs conversions from the original model's axis system and the body axes.



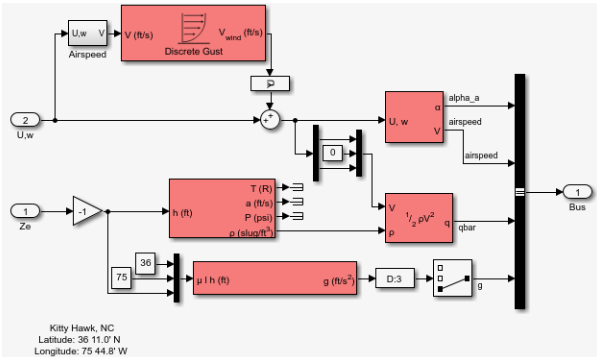
3DOF (Body Axes) Block Parameters

Environment Subsystem

The first and final flights of the Wright Flyer occurred on December 17, 1903. Orville and Wilbur Wright chose an area near Kitty Hawk, North Carolina, situated near the Atlantic coast. Wind gusts of more than 25 miles per hour were recorded that day. After the final flight on that blustery December day, a wind gust caught and overturned the Wright Flyer, damaging it beyond repair.

The Environment subsystem of the Wright Flyer model contains a variety of blocks from the Environment sublibrary of the Aerospace Blockset software, including wind, atmosphere, and gravity, and calculates airspeed and dynamic pressure. The Discrete Wind Gust Model block provides wind gusts to the simulated environment. The other blocks are

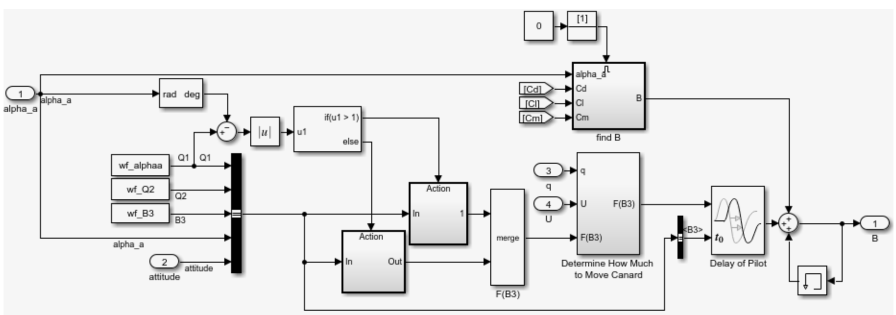
- The Incidence & Airspeed block calculates the angle of attack and airspeed.
- The COESA Atmosphere Model block calculates the air density.
- The Dynamic Pressure block computes the dynamic pressure from the air density and velocity.
- The WGS84 Gravity Model block produces the gravity at the Wright Flyer's latitude, longitude, and height.



Pilot Subsystem

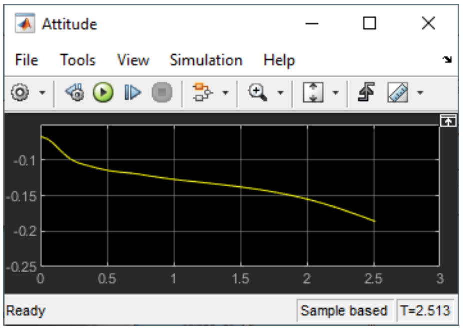
The Pilot subsystem controls the aircraft by responding to both pitch angle (attitude) and angle of attack. If the angle of attack differs from the set angle of attack by more than one degree, the Pilot subsystem responds with a correction of the elevator (canard) angle. When the angular velocity exceeds +/- 0.02 rad/s, angular velocity and angular acceleration are also taken into consideration with additional corrections to the elevator angle.

Pilot reaction time largely determined the success of the flights [1]. Without an automatic controller, a reaction time of 0.06 seconds is optimal for successful flight. The Delay of Pilot (Variable Transport Delay) block recreates this effect by producing a delay of no more than 0.08 second.



Run the Simulation

The default values for this simulation allow the Wright Flyer model to take off and land successfully. The pilot reaction time (`wf_B3`) is set to 0.06 seconds, the desired angle of attack (`wf_alphaa`) is constant, and the altitude attained is low. The Wright Flyer model reacts similarly to the actual Wright Flyer. It leaves the ground, moves forward, and lands on a point as high as that from which it started. This model exhibits the longitudinal undulation in attitude of the original aircraft.



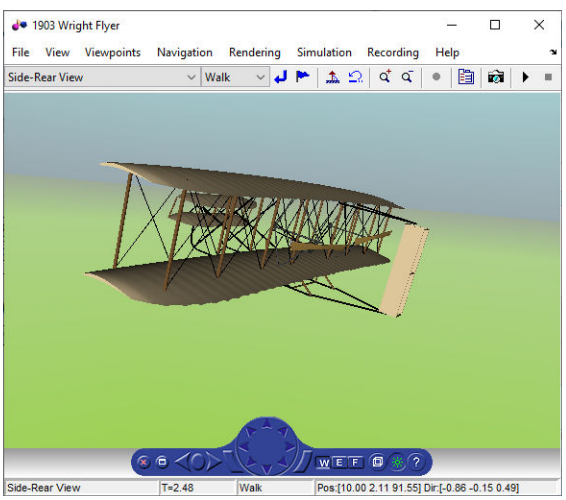
Attitude Scope (Measured in Radians)

A pilot with quick reaction times and ideal flight conditions makes it possible to fly the Wright Flyer successfully. The Wright Flyer model confirms that controlling its longitudinal motion was a serious challenge. The longest recorded flight on that day lasted a mere 59 seconds and covered 852 feet.

Virtual Reality Visualization of the Wright Flyer

Note This section requires the Simulink 3D Animation.

The Wright Flyer model also provides a virtual world visualization, coded in Virtual Reality Modeling Language (VRML) [3]. The VR Sink block in the main model allows you to view the flight motion in three dimensions.



1903 Wright Flyer Virtual Reality World

References

[1] Hooven, Frederick J., "Longitudinal Dynamics of the Wright Brothers' Early Flyers: A Study in Computer Simulation of Flight," from *The Wright Flyer: An Engineering Perspective*, ed. Howard S. Wolko, Smithsonian Institution Press, 1987.

[2] Culick, F. E. C. and H. R. Jex, “Aerodynamics, Stability, and Control of the 1903 Wright Flyer,” from *The Wright Flyer: An Engineering Perspective,* ed. Howard S. Wolko, Smithsonian Institution Press, 1987.

[3] Thaddeus Beier created the initial Wright Flyer model in Inventor format, and Timothy Rohaly converted it to VRML.

See Also

3DOF (Body Axes) | Incidence & Airspeed | COESA Atmosphere Model | Dynamic Pressure | WGS84 Gravity Model

External Websites

- https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/re-living-the-wright-way/

NASA HL-20 Lifting Body Airframe

In this section...

Introduction

This case study models the airframe of a NASA HL-20 lifting body, a low-cost complement to the Space Shuttle orbiter. The HL-20 is unpowered, but the model includes both airframe and controller.

For most flight control designs, the airframe, or plant model, needs to be modeled, simulated, and analyzed. Ideally, this airframe should be modeled quickly, reusing blocks or model structure to reduce validation time and leave more time available for control design. In this study, the Aerospace Blockset software efficiently models portions of the HL-20 airframe. The remaining portions, including calculation of the aerodynamic coefficients, are modeled with the Simulink software. This case study examines the HL-20 airframe model and touches on how the aerodynamic data are used in the model.

NASA HL-20 Lifting Body

The HL-20, also known as the Personnel Launch System (PLS), is a lifting body reentry vehicle designed to complement the Space Shuttle orbiter. It was developed originally as a low-cost solution for getting to and from low Earth orbit. It can carry up to 10 people and a limited cargo [1].

The HL-20 lifting body can be placed in orbit either by launching it vertically with booster rockets or by transporting it in the payload bay of the Space Shuttle orbiter. The HL-20 lifting body deorbits using a small onboard propulsion system. Its reentry profile is nose first, horizontal, and unpowered.

Top-Front View of the HL-20 Lifting Body (Photo: NASA Langley)

The HL-20 design has a number of benefits:

- Rapid turnaround between landing and launch reduces operating costs.
- The HL-20 has exceptional flight safety.
- It can land conventionally on aircraft runways.

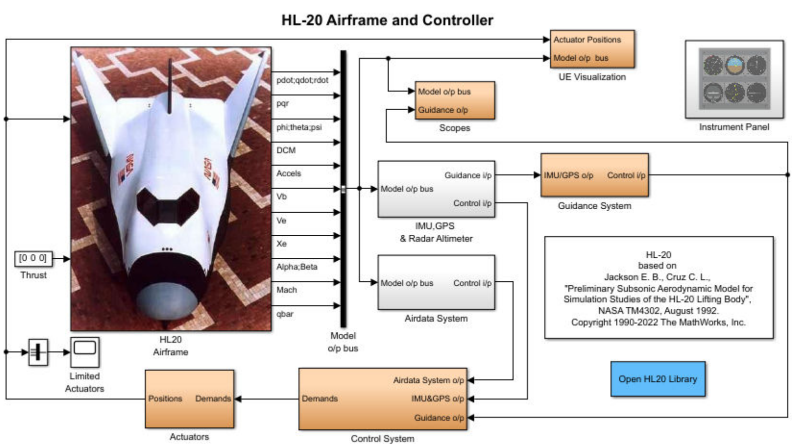
Potential uses for the HL-20 include

- Orbital rescue of stranded astronauts
- International Space Station crew exchanges
- Observation missions
- Satellite servicing missions

Although the HL-20 program is not currently active, the aerodynamic data from HL-20 tests are being used in current NASA projects [2].

The HL-20 Airframe and Controller Model

`Open` the HL-20 airframe and controller model.



Modeling Assumptions and Limitations

Preliminary aerodynamic data for the HL-20 lifting body are taken from NASA document TM4302 [1].

The airframe model incorporates several key assumptions and limitations:

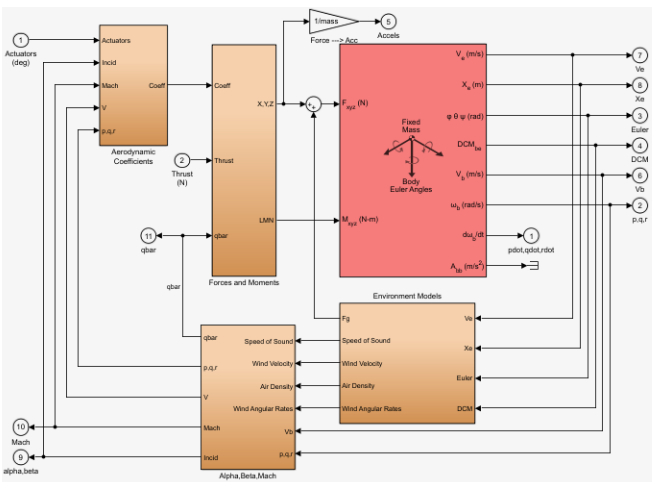
- The airframe is assumed to be rigid and have constant mass, center of gravity, and inertia, since the model represents only the unpowered reentry portion of a mission.
- HL-20 is assumed to be a laterally symmetric vehicle.
- Compressibility (Mach) effects are assumed to be negligible.
- Control effectiveness is assumed to vary nonlinearly with angle of attack and linearly with angle of deflection. Control effectiveness is not dependent on sideslip angle.
- The nonlinear six-degrees-of-freedom aerodynamic model is a representation of an early version of the HL-20. Therefore, the model is not intended for realistic performance simulation of later versions of the HL-20.

The typical airframe model consists of a number of components, such as

- Equations of motion
- Environmental models
- Calculation of aerodynamic coefficients, forces, and moments

The airframe subsystem of the HL-20 model contains five subsystems, which model the typical airframe components:

- “6DOF (Euler Angles) Subsystem” on page 3-16
- “Environmental Models Subsystem” on page 3-16
- “Alpha, Beta, Mach Subsystem” on page 3-18
- “Aerodynamic Coefficients Subsystem” on page 3-19
- “Forces and Moments Subsystem” on page 3-21



HL-20 Airframe Subsystem

6DOF (Euler Angles) Subsystem

The 6DOF (Euler Angles) subsystem contains the six-degrees-of-freedom equations of motion for the airframe. In the 6DOF (Euler Angles) subsystem, the body attitude is propagated in time using an Euler angle representation. This subsystem is one of the equations of motion blocks from the Aerospace Blockset library. A quaternion representation is also available. See the 6DOF (Euler Angles) and 6DOF (Quaternion) block reference pages for more information on these blocks.

Environmental Models Subsystem

The Environmental Models subsystem contains the following subsystems and blocks:

- The WGS84 Gravity Model block implements the mathematical representation of the geocentric equipotential ellipsoid of the World Geodetic System (WGS84).

  See the WGS84 Gravity Model block reference page for more information on this block.
- The COESA Atmosphere Model block implements the mathematical representation of the 1976 Committee on Extension to the Standard Atmosphere (COESA) standard lower atmospheric values for absolute temperature, pressure, density, and speed of sound, given the input geopotential altitude.

See the COESA Atmosphere Model block reference page for more information on this block.

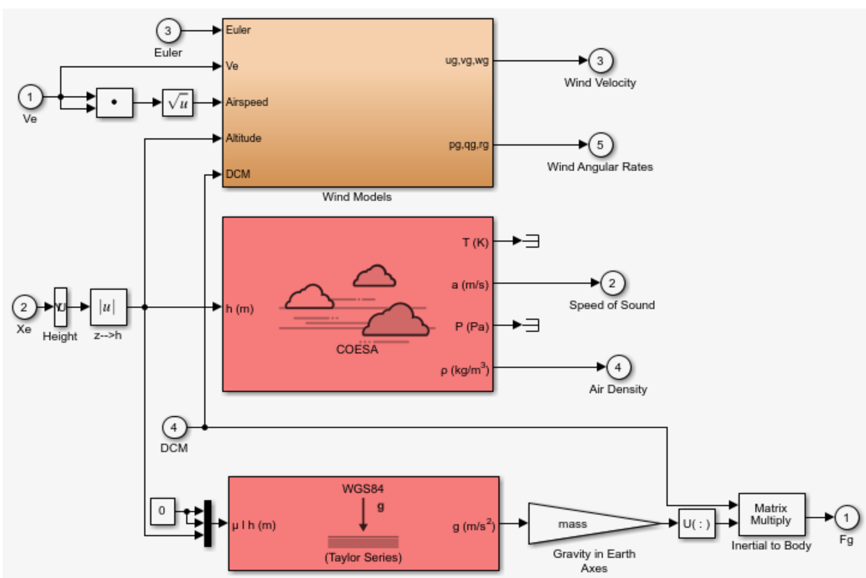
- The Wind Models subsystem contains the following blocks:
  - The Wind Shear Model block adds wind shear to the model.

    See the Wind Shear Model block reference page for more information on this block.
  - The Discrete Wind Gust Model block implements a wind gust of the standard “1 - cosine” shape.

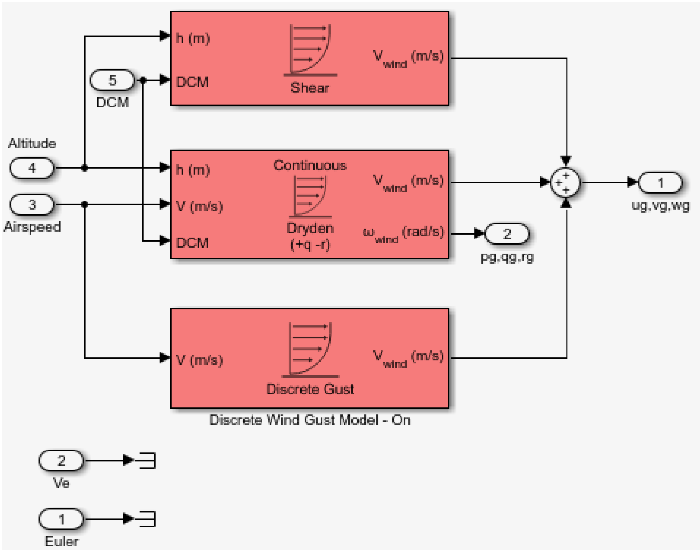
    See the Discrete Wind Gust Model block reference page for more information on this block.
  - The Dryden Wind Turbulence Model (Continuous) block uses the Dryden spectral representation to add turbulence to the aerospace model by passing band-limited white noise through appropriate forming filters.

    See the Dryden Wind Turbulence Model (Continuous) block reference page for more information on this block.

The environmental models implement mathematical representations within standard references, such as U.S. Standard Atmosphere, 1976.



Environmental Models in HL-20 Airframe Model



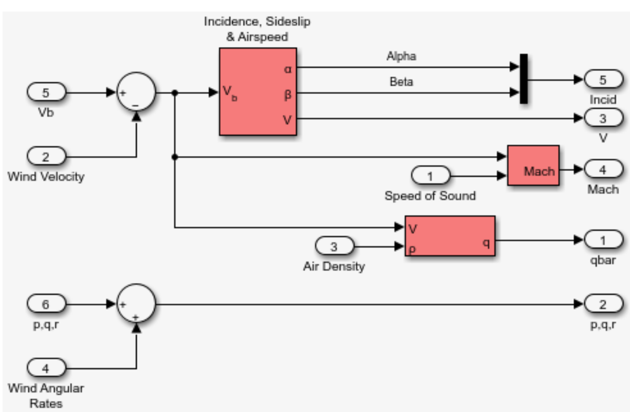
Wind Models in HL-20 Airframe Model

Alpha, Beta, Mach Subsystem

The Alpha, Beta, Mach subsystem calculates additional parameters needed for the aerodynamic coefficient computation and lookup. These additional parameters include

- Mach number
- Incidence angles (α, β)
- Airspeed
- Dynamic pressure

The Alpha, Beta, Mach subsystem corrects the body velocity for wind velocity and corrects the body rates for wind angular acceleration.



Additional Computed Parameters for HL-20 Airframe Model (Alpha, Beta, Mach Subsystem)

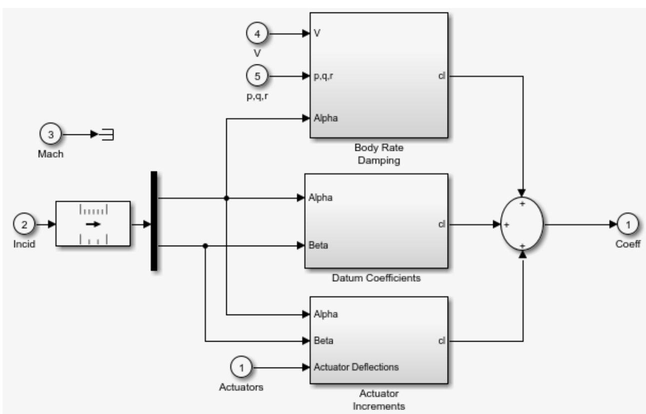
Aerodynamic Coefficients Subsystem

The Aerodynamic Coefficients subsystem contains aerodynamic data and equations for calculating the six aerodynamic coefficients, which are implemented as in reference [1]. The six aerodynamic coefficients follow.

C_x	Axial-force coefficient
C_y	Side-force coefficient
C_z	Normal-force coefficient
C_l	Rolling-moment coefficient
C_m	Pitching-moment coefficient
C_n	Yawing-moment coefficient

Ground and landing gear effects are not included in this model.

The contribution of each of these coefficients is calculated in the subsystems (body rate, actuator increment, and datum), and then summed and passed to the Forces and Moments subsystem.



Aerodynamic Coefficients in HL-20 Airframe Model

The aerodynamic data was gathered from wind tunnel tests, mainly on scaled models of a preliminary subsonic aerodynamic model of the HL-20. The data was curve fitted, and most of the aerodynamic coefficients are described by polynomial functions of angle of attack and sideslip angle. In-depth details about the aerodynamic data and the data reduction can be found in reference [1].

The polynomial functions contained in the `aeroblk_init_hl20.m` file are used to calculate lookup tables used by the model's preload function. Lookup tables substitute for polynomial functions. Depending on the order and implementation of the function, using lookup tables can be more efficient than recalculating values at each time step with functions. To further improve efficiency, most tables are implemented as PreLook-up Index Search and Interpolation (n-D) using PreLook-up blocks. These blocks improve performance most when the model has a number of tables with identical breakpoints. These blocks reduce the number of times the model has to search for a breakpoint in a given time step. Once the tables are populated by the preload function, the aerodynamic coefficient can be computed.

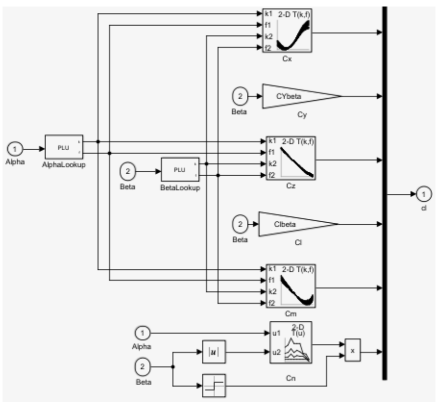
The equations for calculating the six aerodynamic coefficients are divided among three subsystems:

- “Datum Coefficients Subsystem” on page 3-20
- “Body Rate Damping Subsystem” on page 3-20
- “Actuator Increment Subsystem” on page 3-20

Summing the Datum Coefficients, Body Rate Damping, and Actuator Increments subsystem outputs generates the six aerodynamic coefficients used to calculate the airframe forces and moments [1].

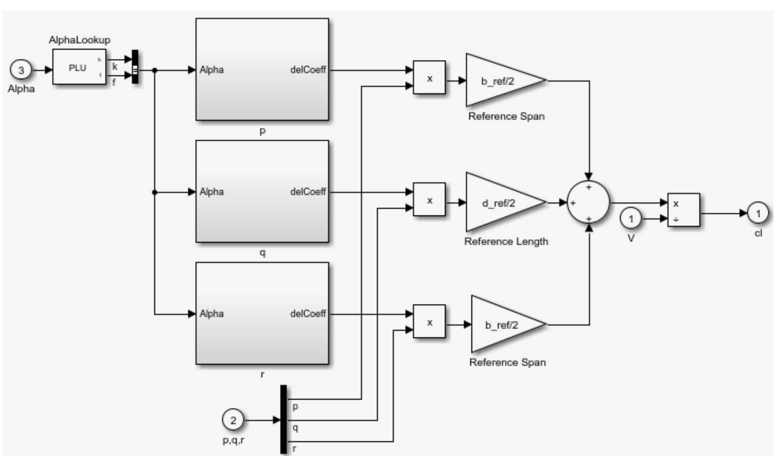
Datum Coefficients Subsystem

The Datum Coefficients subsystem calculates coefficients for the basic configuration without control surface deflection. These datum coefficients depend only on the incidence angles of the body.



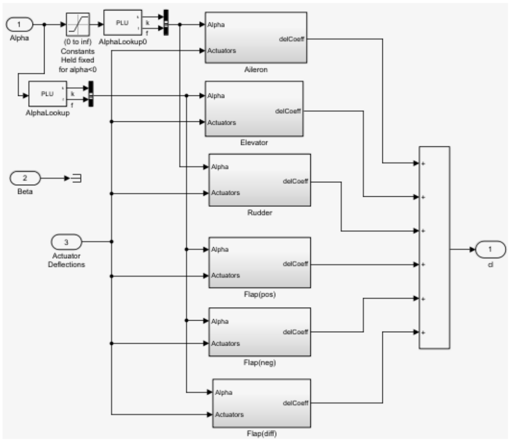
Body Rate Damping Subsystem

Dynamic motion derivatives are computed in the Body Rate Damping subsystem.



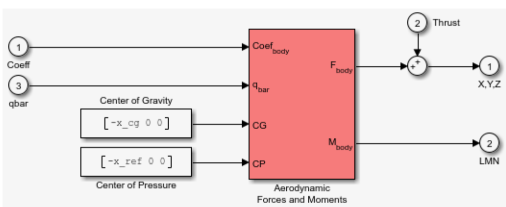
Actuator Increment Subsystem

Lookup tables determine the incremental changes to the coefficients due to the control surface deflections in the Actuator Increment subsystem. Available control surfaces include symmetric wing flaps (elevator), differential wing flaps (ailerons), positive body flaps, negative body flaps, differential body flaps, and an all-movable rudder.



Forces and Moments Subsystem

The Forces and Moments subsystem calculates the body forces and body moments acting on the airframe about the center of gravity. These forces and moments depend on the aerodynamic coefficients, thrust, dynamic pressure, and reference airframe parameters.



Complete the Model

These subsystems that you have examined complete the HL-20 airframe. The next step in the flight control design process is to analyze, trim, and linearize the HL-20 airframe so that a flight control system can be designed for it. To see an example of an auto-land flight control for the HL-20 airframe, type the command `openExample(aeroblk_HL20_UE)`, at the MATLAB command line.

References

[1] Jackson, E. B., and C. L. Cruz, “Preliminary Subsonic Aerodynamic Model for Simulation Studies of the HL-20 Lifting Body,” NASA TM4302 (August 1992)..

[2] Morring, F., Jr., “ISS `Lifeboat' Study Includes ELVs,” *Aviation Week & Space Technology* (May 20, 2002).

See Also

External Websites

- http://www.astronautix.com/h/hl-20.html

Model a Quadcopter Based on Parrot Minidrones

This workflow shows how to use Simulink to model a quadcopter based on the Parrot series of minidrones.

- The steps in this workflow describe the implementation details of the Parrot minidrone used in “Quadcopter Modeling and Simulation based on Parrot Minidrone” on page 9-81.
- This workflow shows how to manage the model and source files, using “Project Management”.
- This workflow shows the quadcopter in a 3D environment, using Simulink 3D Animation.
- For the collaborative development of a flight simulation application, this workflow provides an implementation of Flight Simulation template. For more information, see “Projects Template for Flight Simulation Applications” on page 2-54.

For hardware integration, this procedure requires the installation of the Simulink Support Package for Parrot Minidrones.

1. Follow the steps in “Quadcopter Modeling and Simulation based on Parrot Minidrone” on page 9-81 to open a Simulink project with the `asbQuadcopter` model and load the required workspace variables.
2. Run the `asbQuadcopter` model in normal simulation mode. Navigate through the different subsystems to learn about the modeling hierarchy and quadcopter dynamics. You can also view the Simulink 3D Animation of the model. For more information, see “Quadcopter Modeling and Simulation based on Parrot Minidrone” on page 9-81.

Each aspect of the quadcopter implementation details are described in these steps:

1. “Quadcopter Physical Characteristics” on page 3-24
2. “Quadcopter Control and Sensors” on page 3-26
3. “Quadcopter Airframe” on page 3-29
4. “Quadcopter Project Environment” on page 3-31
5. “Quadcopter Visualization” on page 3-33

See Also

Related Examples

- “Quadcopter Modeling and Simulation based on Parrot Minidrone” on page 9-81

Quadcopter Physical Characteristics

This schematic shows these quadcopter physical characteristics:

- Axes
- Mass and Inertia
- Rotors

Axes

The quadcopter body axis is centered in the center of gravity.

- **X-Axis** — The *X*-axis is represents the longitudinal axis of the drone, extending from the front to the rear of the drone. Movement along the *X*-axis involves pitching the drone forward or backward.
- **Y-Axis** — The *Y*-axis represents the transverse axis of the drone, extending from one side of the drone to the other. Movement along the *Y*-axis involves rolling the drone to the left or right.
- **Z-Axis** — The *Z*-axis represents the vertical axis of the drone, extending from the top to the bottom of the drone, following the right-hand rule. Movement along the *Z*-axis involves ascending or descending, as well as yawing the drone to change its heading.

These axes are crucial for understanding and controlling the movement and orientation of the drone in three-dimensional space. They are fundamental for drone control, navigation, and stabilization.

Mass and Inertia

Assume that the whole body works as a rigid body of uniform density. The file `vehicleVars` contains the values for the inertia and mass.

Rotors

The Parrot quadcopter minidrone body typically consists of four rotors, often referred to as propellers. These rotors are responsible for generating the lift required for the drone to achieve flight. In most quadcopters, including the Parrot minidrone, the rotors are configured in a cross pattern, with two rotors spinning clockwise and the other two spinning counterclockwise.

This configuration provides stability and control, as the opposing rotation cancels out the rotational torque, ensuring the drone remains level during flight. Additionally, you can independently control the speed of each rotor to enable the drone to change altitude, pitch, roll, and yaw.

The rotor system is a critical component of the design of the drone. The rotor system is key to maneuverability, stability, and overall flight performance.

- Rotor 1 rotates positively with respect to the z-axis. This rotor is located parallel to the *xy*-plane, -45 degrees from the *x*-axis.
- Rotor 2 rotates negatively with respect to the *z*-axis of the body. This rotor is located parallel to the *xy*-plane, 45 degrees from the *x*-axis.
- Rotor 3 has the same rotation direction as rotor 1. This rotor is located parallel to the *xy*-plane, 135 degrees from the *x*-axis.
- Rotor 4 has the same rotation direction as rotor 2. This rotor is located parallel to the *xy*-plane, -135 degrees from the *x*-axis.

This example uses the Multirotor block in the Aerospace Blockset. The block is based on the approach defined by Prouty and adapted to a heavy-lift quadcopter by Pounds , Mahony, and Corke [1].

Command Subsystem

Use one of these options to provide inputs to the quadcopter for pitch (*X*), roll (*Y*), and yaw (*Z*), as north-east-down (NED) coordinates:

- A Signal Editor block
- A joystick
- Previously saved data
- A spreadsheet

Use the `VSS_COMMAND` variable in the workspace to make the appropriate choice. This subsytem generates reference command signal to the control system.

References

[1] Pounds, P., R. Mahony, and P. Corke. "Modelling and Control of a Large Quadrotor Robot." *Control Engineering Practice* 18, no. 7 (2010): 691–99. https://doi.org/10.1016/j.conengprac.2010.02.008.

Quadcopter Control and Sensors

You must create a control system and set sensors for the quadcopter.

Control System

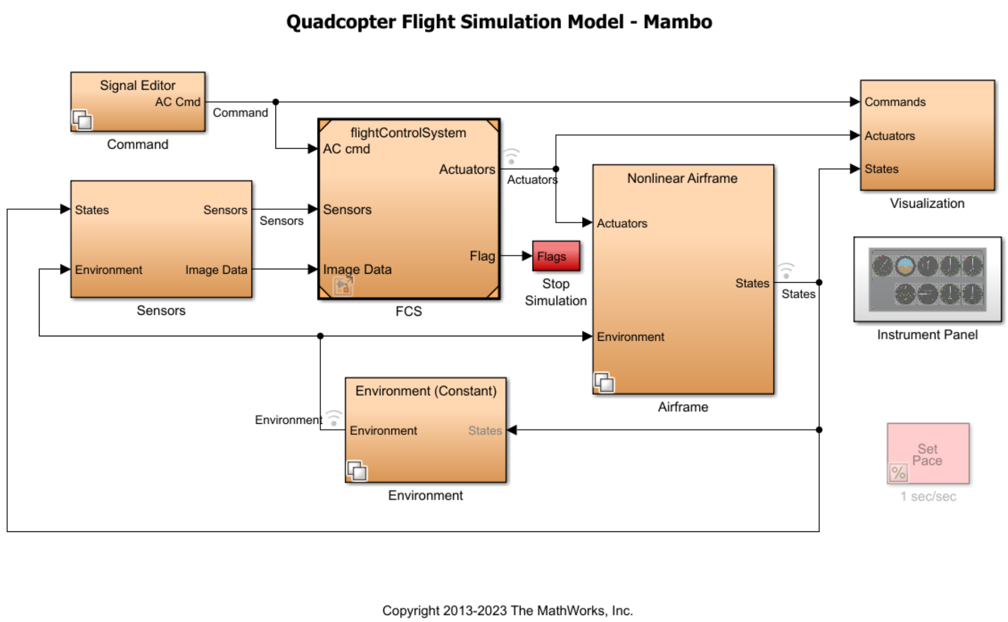
To create a control system for a Parrot quadcopter minidrone , you must design algorithms and controllers that manage the stability, altitude, orientation, and trajectory of the drone.

When designing the control system for a minidrone, consider the specific dynamics and constraints of the drone, as well as the requirements of the intended mission or application. Safety, stability, and responsiveness are critical aspects of the control system design.

For control, the quadcopter uses a complementary filter to estimate attitude, and Kalman filters to estimate position and velocity.

The model implements the controller and estimators as subsystems, which allows you to evaluate several combinations of estimators and controllers for design.

Quadcopter Flight Control System



Inputs

The inputs of the flight control system (FCS) are:

- Aircraft command — From `Command` subsystem
- Sensors — From the `Sensor` subsystem
- Image data — From `Sensors` subsystem

Components

The main components of the FCS are:

1. *Image Processing system* — The system uses the image data from the sensors and computes the landing flag, which forms the input to the landing logic control system. The subsystem detects whether dominance of red is detected in the image from the camera.
2. *Landing Logic* — The landing logic system initiates the controller to land the quadcopter based on the landing flag, which is set based on the camera data, and the landing Override flag, which is set if the vehicle closes in on the safe limit of 0.6 m (height from the ground).
3. *Estimators* — The subsystem uses the sensor subsystem outputs to estimate the states of the vehicle using a complementary filter (for attitude) and Kalman filters (for position and velocity).
4. *Controller* — The implemented controller logic uses

   - A PID controller for pitch/roll control.
   - A PD controller for yaw
   - A PD controller for position control in North-East-Down coordinates
5. *Crash predictor flag* — The flag is set based on the vehicle state values. The simulation is stopped based on the optical flow velocity (u or v is >0 .01) or the estimated position (x or y >10).

The `controllerVars` file contains variables pertinent to the controller. The `estimatorVars` file contains variables pertinent to the estimator. The values are based on these files https://github.com/Parrot-Developers/RollingSpiderEdu/tree/master/MIT_MatlabToolbox/trunk/matlab/libs/RoboticsToolbox.

Sensors

You can use either of these types of sensors in the model:

- Dynamic sensors
- Feedthrough sensors

Dynamic sensors and feedthrough sensors serve different purposes and provide distinct types of data for the flight control system. Dynamic sensors are designed to measure changes in the motion and orientation of the drone. They provide real-time data related to the movement, acceleration, and rotational dynamics of the drone. Common dynamic sensors in a drone include accelerometers, gyroscopes, magnetometers, and inertial measurement units (IMUs).

Feedthrough sensors directly pass on the states and environment values as sensor outputs, while in Dynamic sensors the IMU block adds noise to the state values.

Use the `VSS_SENSORS` variable in the workspace to choose your sensors.

This workflow uses a set of sensors to determine the states of the quadcopter. These states describe the orientation and position of the quadcopter:

- V_{NED} — Velocity in earth reference frame
- X_{NED} — Position in earth reference frame
- *Euler* — Euler rotation angles φ, θ, and ψ
- DCM_{be} — Coordinate transformation from Earth axes to body-fixed axes

- V_b — Velocity in the body-fixed frame
- ω_b — Angular rates in body-fixed axes
- $d\omega_b/dt$ — Angular accelerations
- A_{be} — Accelerations with respect to inertial frame

These sensors determine the states:

- An IMU to measure the angular rates and translational accelerations.
- An algorithm for optical flow estimation.
- Sonar for altitude measurement.

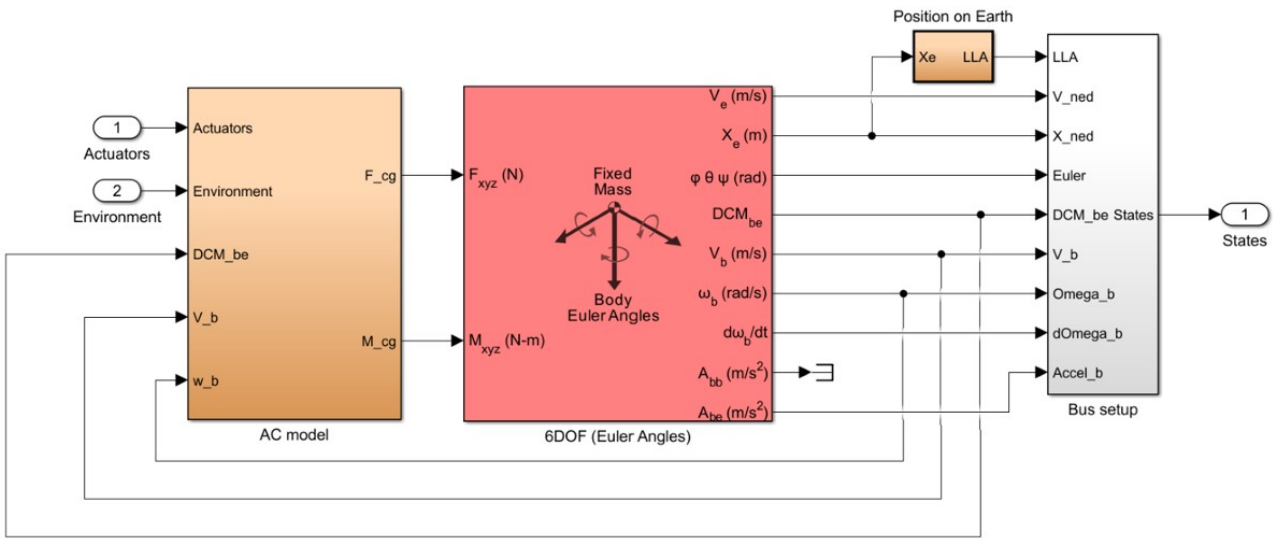
The system stores the characteristics for the sensors in `sensorVars`. To include sensor dynamics with these measurements, you can change the `VSS_SENSORS` variable in the workspace.

Quadcopter Airframe

You can use one of these two approaches to implement the airframe model.

Nonlinear Airframe

For a nonlinear airframe, the model computes gravity forces, profile drag, and the aerodynamic forces and moments.



The `AC Model` subsystem computes the total force using these equations:

Gravity: $F = mg$, where m is the quadcopter mass, and g is the acceleration due to gravity.

Profile drag: $D = \frac{1}{2}\rho V^2 S C_d$, where ρ is the atmospheric density, V is the body velocity, S is the reference area, and C_d is the drag coefficient.

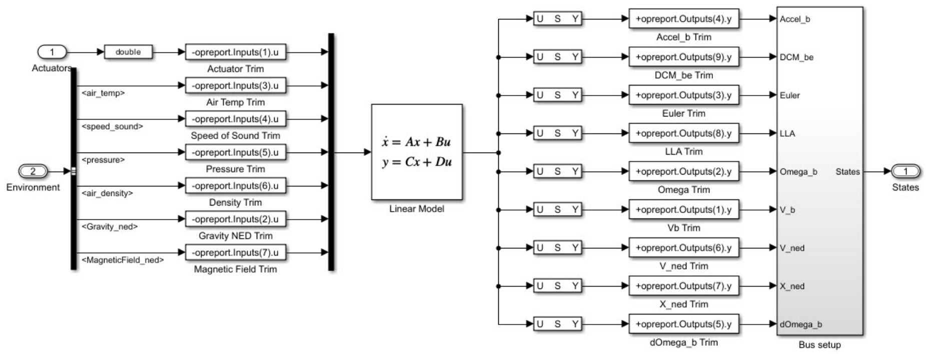
Aerodynamic forces and moments: Computed using the Multirotor block with the inclusion of flap effects.

The 6DOF (Quaternion) block integrates the equations of motion of the vehicle to obtain the states at each time instant.

The nonlinear workflow accurately represents the dynamic behavior of the drones in a three-dimensional space. The 6DOF (Quaternion) block represents the orientation and position of the drone in a 3D environment using quaternions, providing a more robust representation of its motion and orientation.

Linear Airframe

For a linear airframe, the trim values for the actuator inputs, the environment parameter values, and the state space A,B,C,D matrices for the linearized model are saved in the MAT file `linearizedAirframe`. These values are calculated using the `trimLinearizeOpPoint` function, which linearizes the nonlinear model of the quadcopter using Simulink Control Design.



Quadcopter Project Environment

Environment

The environment in which a Parrot quadcopter minidrone operates can vary widely, ranging from indoor settings to outdoor environments. Understanding the environment is crucial for safe and effective drone operation.

The environmental parameters considered in this workflow are:

1. **Gravity** — Gravity plays a fundamental role in the flight dynamics of the drone. It affects the weight, lift, and overall stability of the drone during flight. The force of gravity influences the ability of the drone to ascend, descend, and maintain altitude, as well as its overall maneuverability.
2. **Air temperature** — Air temperature affects the air density, which in turn impacts the aerodynamic performance of the drone. As temperature increases, air density decreases, affecting lift and drag forces on the drone. Additionally, temperature changes can influence the efficiency of the propulsion system of the drone.
3. **Speed of sound** — The speed of sound in the surrounding air affects the propagation of acoustic signals, including those used for communication and telemetry. Changes in the speed of sound due to temperature and pressure variations can impact the acoustic sensors of the drone and communication systems.
4. **Pressure** — Air pressure influences the aerodynamic forces acting on the drone. Changes in air pressure affect lift, drag, and overall aerodynamic performance. Additionally, variations in air pressure can impact altitude measurements and barometric sensors used for altitude hold and navigation.
5. **Air density** — Air density directly affects the aerodynamic performance of the drone. Lower air density reduces lift and increases the required airspeed for the same amount of lift. Changes in air density due to altitude or temperature variations influence the flight characteristics and performance of the drone.
6. **Magnetic field** — The magnetic field of the Earth affects the onboard magnetometer of the drone, which is used for orientation and navigation. Variations in the magnetic field due to geographic location and local magnetic anomalies can impact the accuracy of the compass of the drone and its ability to maintain heading and orientation.

Two options are available for the implementation of the environment models.

- Constant — In the constant environment model, the environmental parameters are fixed at their standard sea-level values.
- Variable — In the variable environment model, the latitude, longitude, and altitude (LLA) coordinates of the vehicle obtained in States from the `Airframe` subsystem are used as input to different environment blocks from the Aerospace Blockset to obtain the different atmospheric parameters.
  - The WGS84 gravity model is used to compute the gravity of the Earth at a specific location.
  - The COESA Atmosphere model is used to obtain the temperature, speed of sound, pressure and density of air based on the altitude input.
  - The World Magnetic Model 2015 is used to calculate the magnetic field of the Earth at a specific location and time.

To include these models, you can change the `VSS_ENVIRONMENT` variable in the workspace to toggle between variable and constant environment models.

Quadcopter Visualization

You can visualize the variables for the quadcopter in one of these ways:

- Use the Simulation Data Inspector.
- Use the “Flight Instruments” blocks.
- Toggle between the different visualization variant subsystems.

  You can toggle between the different variant subsystems by changing the `VSS_VISUALIZATION` variable. These options include:

  - Scope
  - Workspace
  - Simulink 3D Animation — Airport and Apple Hill scenes. The Apple Hill scene corresponds to the `VSS_VISUALIZATION = 4` variable.
  - “Flight Simulator Interfaces” — This variant is a FlightGear animation. To use this animation, add a FlightGear-compatible model of the quadcopter to the project. The software does not include this model.

The may also use Aerospace Blockset uses Unreal Engine for 3D visualization. For details on installing the support package and using customized scenes, see “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2.

To fly in the Apple Hill or a custom scene:

- Double-click the Simulation 3D Scene Configuration block in to open its mask.
- Enter the project location where you saved the `AutoVrtlEnv.uproject` file from the support package, then click the **Open Unreal Editor**.
- To save your changes and close the mask, click **OK**.
- In Unreal Editor, change the map to **AppleHill** by finding the folder `MathWorksAerospaceContent`. Click **Content** > **Maps** and select **AppleHill**.

Supporting Data

Customize 3D Scenes for Aerospace Blockset Simulations

Aerospace Blockset contains a prebuilt airport scene in which to simulate and visualize the performance of aerospace vehicles modeled in Simulink. This scene is visualized using a standalone Unreal executable within the toolbox. If you have Unreal from Epic Games and the Aerospace Blockset Interface for Unreal Engine Projects installed, you can customize this scene as well as an additional Griffiss International Airport scene. You can also use the support package to simulate within your scenes from your own custom project.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

With custom scenes, you can co-simulate in both Simulink and the Unreal Editor so that you can modify your scenes between simulation runs. To customize scenes, you should be familiar with creating and modifying scenes in the Unreal Editor.

To customize 3D scenes, follow these steps:

1. “Install Support Package and Configure Environment” on page 4-3
2. “Migrate Projects Developed Using Prior Support Packages” on page 4-5
3. “Customize Scenes Using Simulink and Unreal Editor” on page 4-7
4. “Package Custom Scenes into Executable” on page 4-15

See Also

Simulation 3D Scene Configuration

Related Examples

- Using Unreal Engine with Simulink Video Tutorial
- “Get Started Communicating with the Unreal Engine Visualization Environment” on page 4-18
- “Create Empty Project in Unreal Engine” on page 4-51
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37

External Websites

- Unreal Engine
- Unreal Engine 5 Documentation

Install Support Package and Configure Environment

To customize scenes in your installation of the Unreal Editor and simulate within these scenes in Simulink, you must first install and configure the Aerospace Blockset Interface for Unreal Engine Projects support package.

Verify Software and Hardware Requirements

Before installing the support package, make sure that your installed version of Unreal and your environment meets the requirements described in “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37. This topic also lists the supported Unreal Engine version.

Note These installation instructions apply to R2023a. If you are using a previous release, see the documentation for Other Releases.

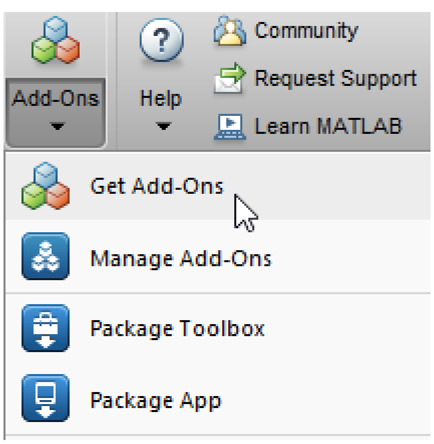
Verify Software and Hardware Requirements

Before installing the support package, make sure that your environment meets the minimum software and hardware requirements described in “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37.

Install Support Package

To install the Aerospace Blockset Interface for Unreal Engine Projects support package, follow these steps:

1. On the MATLAB **Home** tab, in the **Environment** section, select **Add-Ons** > **Get Add-Ons**.



2. In the Add-On Explorer window, search for the Aerospace Blockset Interface for Unreal Engine Projects support package. Click **Install**.

Note You must have write permission for the installation folder.

Configure Environment

The Aerospace Blockset Interface for Unreal Engine Projects support package includes these components:

- An Unreal project, `AutoVrtlEnv.uproject`, and its associated files. The project includes editable versions of the prebuilt 3D scenes that you can select from the **Scene description** parameter of the Simulation 3D Scene Configuration block. To use this project, you must copy the file to a folder on your local machine.
- A plugin, `MathWorkInterface` (with a folder name of `MathWorkSimulation`). This plugin establishes the connection between MATLAB and the Unreal Editor and is required for co-simulation. It also includes some shared automotive-oriented assets. You must copy this plugin to your local installation of the editor.
- A second plugin, `MathWorksAerospaceContent`. This plugin contains the aerospace components and connects them to MATLAB using the `MathWorksSimulation` plugin. You must also copy this plugin to your local installation of the editor.
- A third plugin, `RoadRunnerMaterials`. This plugin is required for scenes created by the RoadRunner scene editing software, and for packaging the project into an executable.

To configure your environment so that you can customize scenes, use `copyExampleSim3dProject` to copy the support package components to a folder on your local machine. For example, this code copies the files to `C:\project`.

```
sim3d.utils.copyExampleSim3dProject("C:\project");
```

If you want to use a project developed using a prior release of the Aerospace Blockset Interface for Unreal Engine Projects support package, you must migrate the project to make it compatible with Unreal Editor 4.27. See “Migrate Projects Developed Using Prior Support Packages” on page 4-5. Otherwise, you can “Customize Scenes Using Simulink and Unreal Editor” on page 4-7.

Note If you want to use the plugins to co-simulate with more than one Unreal project, see Unreal Engine 4.27 Plugins.

See Also

Simulation 3D Scene Configuration | `copyExampleSim3dProject`

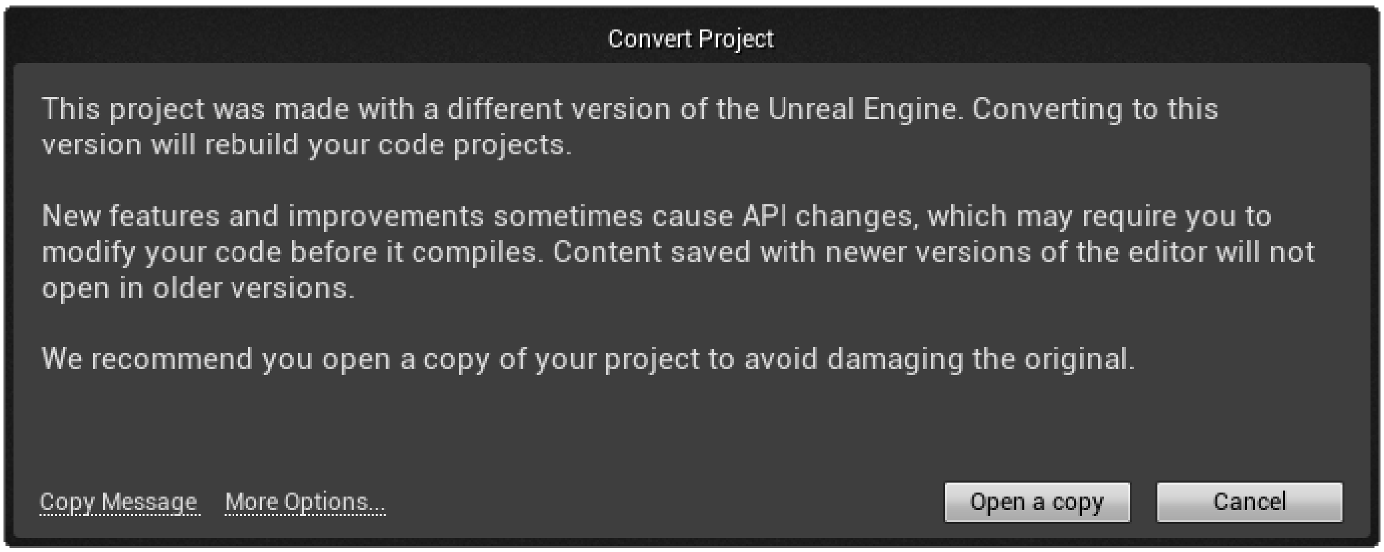
More About

- “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2

Migrate Projects Developed Using Prior Support Packages

After you install the Aerospace Blockset Interface for Unreal Engine Projects support package as described in "Install Support Package and Configure Environment" on page 4-3, you may need to migrate your project. If your Simulink model uses an Unreal Engine executable or project developed using a prior release of the support package, you must migrate the project to make it compatible with Unreal Editor 5.1. Follow these steps:

1. Open Unreal Engine 5.1. For example, navigate to `C:\Program Files\Epic Games\UE_5.1\Engine\Binaries\Win64` and open `UnrealEditor.exe`.
2. Use the Unreal Project Browser to open the project that you want to migrate.
3. Follow the prompts to open a copy of the project. The editor creates a new project folder in the same location as the original, appended with 5.1. Close the editor.



4. In a file explorer, remove any spaces in the migrated project folder name. For example, rename `MyProject 5.1` to `MyProject5.1`.
5. Use MATLAB to open the migrated project in Unreal Editor 5.1. For example, if you have a migrated project saved to the `C:/Local` folder, use this MATLAB code:

```
path = fullfile('C:','Local','MyProject5.1','MyProject.uproject');
editor = sim3d.Editor(path);
open(editor);
```

Note The support package may includes changes in the implementation of some actors. Therefore, if the original project contains actors that are placed in the scene, some of them might not fully migrate to Unreal Editor 5.1. To check, examine the Output Log.

The log might contain error messages. For more information, see the Unreal Engine Documentation or contact MathWorks Technical Support.

---

6. Optionally, after you migrate the project, you can use the project to create an Unreal Engine executable. See “Package Custom Scenes into Executable” on page 4-15.

After you migrate the project, you can create custom scenes. See “Customize Scenes Using Simulink and Unreal Editor” on page 4-7.

Tip If your project cannot locate the support package plugins, you may need to copy the plugins to the Unreal plugin folder or the Unreal project folder.

See Also
Simulation 3D Scene Configuration

More About
- “Customize Scenes Using Simulink and Unreal Editor” on page 4-7

Customize Scenes Using Simulink and Unreal Editor

After you install the Aerospace Blockset Interface for Unreal Engine Projects support package as described in “Install Support Package and Configure Environment” on page 4-3, you can simulate in custom scenes simultaneously from both the Unreal Editor and Simulink. By using this co-simulation framework, you can add aircraft and sensors to a Simulink model and then run this simulation in your custom scene.

To use a project that you developed using a prior release of the support package, first migrate the project to the currently supported Unreal Engine. See “Migrate Projects Developed Using Prior Support Packages” on page 4-5.

Open Unreal Editor

Simulink will fail to establish a connection with the editor if you open the Unreal Editor outside MATLAB or Simulink. To establish this connection, you must open your project from a Simulink model or use a MATLAB function.

The first time that you open the Unreal Editor, you might be asked to rebuild `UE4Editor` DLL files or the `AutoVrtlEnv` module. Click **Yes** to rebuild these files or modules. The editor also prompts you that new plugins are available. Click **Manage Plugins** and verify that the **MathWorks Interface** plugin is installed. This plugin is the `MathWorksSimulation.uplugin` file that you copied into your Unreal Editor installation in “Install Support Package and Configure Environment” on page 4-3.

Messages about files with the name `'_BuiltData'` indicate missing lighting data for the associated level. You should rebuild these levels' lighting before shipping an executable

If you receive a warning that the lighting needs to be rebuilt, from the toolbar above the editor window, select **Build** > **Build Lighting Only**. The editor issues this warning the first time you open a scene or when you add new elements to a scene. To use the lighting that comes installed with `AutoVrtlEnv` in Aerospace Blockset, see “Use AutoVrtlEnv Project Lighting in Custom Scene” on page 4-10.

Open Unreal Editor from Simulink

1. Open a Simulink model configured to simulate in the 3D environment. At a minimum, the model must contain a Simulation 3D Scene Configuration block.
2. In the Simulation 3D Scene Configuration block of this model, set the **Scene source** parameter to `Unreal Editor`.
3. In the **Project** parameter, browse for the project file that contains the scenes that you want to customize.

   For example, this sample path specifies the `AutoVrtlEnv` project that comes installed with the Aerospace Blockset Interface for Unreal Engine Projects support package.

   ```
   C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject
   ```

   This sample path specifies a custom project.

   ```
   Z:\UnrealProjects\myProject\myProject.uproject
   ```
4. Click **Open Unreal Editor**. The Unreal Editor opens and loads a scene from your project.

Open Unreal Editor Using Command-Line Function

To open the `AutoVrtlEnv.uproject` file that was copied from the Aerospace Blockset Interface for Unreal Engine Projects support package, specify the path to where you copied this project. For example, if you copied the `AutoVrtlEnv.uproject` to `C:/Local/AutoVrtlEnv`, use this code:

```
path = fullfile('C:','Local','AutoVrtlEnv','AutoVrtlEnv.uproject');
editor = sim3d.Editor(path);
open(editor);
```

The editor opens the `AutoVrtlEnv.uproject` file.

To open your own project, use the same commands used to open the `AutoVrtlEnv.uproject` file. Update the `path` variable with the path to your `.uproject` file. For example, if you have a project saved to the `C:/Local` folder, use this code:

```
path = fullfile('C:','Local','myProject','myProject.uproject');
editor = sim3d.Editor(path);
open(editor);
```

Create or Modify Scenes in Unreal Editor

After you open the editor, you can modify the scenes in your project or create new scenes.

Open Scene

In the Unreal Editor, scenes within a project are referred to as levels. Levels come in several types, and scenes have a level type of map.

To open a prebuilt scene from the `AutoVrtlEnv.uproject` file, in the **Content Browser** pane below the editor window, navigate to the **MathWorksAerospaceContent > Maps** folder. Then, double-click the map that corresponds to the scene you want to modify.

Note If the tree does not appear in the **Content Browser**, click the three-line icon in the top right corner of the browser. If **MathWorksAerospaceContent** does not appear in the tree, in the lower right corner of the browser, click **View Options** and select **Show Engine Content** and **Show Plugin Content**.

Unreal Editor Map	Aerospace Blockset Scene
`Airport`	Airport
`GriffissAirport`	Griffiss International Airport

To open a scene within your own project, in the **Content Browser** pane, navigate to the folder that contains your scenes.

Send Data to Scene

The Simulation 3D Message Get block retrieves data from the Unreal Engine 3D visualization environment. To use the block, you must configure scenes in the Unreal Engine environment to send data to the Simulink model.

Receive Data from Scene

The Simulation 3D Message Set block sends data to the Unreal Engine 3D visualization environment. To use the block, you must configure scenes in the Unreal Engine environment to receive data from the Simulink model.

Create New Scene

To create a new scene in your project, from the top-left menu of the editor, select **File > New Level**.

Alternatively, you can create a new scene from an existing one. This technique is useful if you want to use one of the prebuilt scenes in the `AutoVtrlEnv` project as a starting point for creating your own scene. To save a version of the currently opened scene to your project, from the top-left menu of the editor, select **File > Save Current As**. The new scene is saved to the same location as the existing scene.

Add Assets to Scene

In the Unreal Editor, elements within a scene are referred to as assets. To add assets to a scene, you can browse or search for them in the **Content Browser** pane at the bottom and drag them into the editor window.

When adding assets to a scene it is helpful to understand the origin and orientation of that scene’s coordinate system.

Map	Coordinate Direction	Origin and Orientation
Airport	*X*	Aligned with the runway, *X* = 3532.5 meters at the start of the runway.
	Y	*Y* = 0 at runway center.
	Z	*Z* = 0 at ground level.
Griffiss Airport	*X*	Aligned with true North, *X* = 200.1 meters at the threshold of runway 33.
	Y	Aligned with true East, *Y* = -194.7 meters at the threshold of runway 33.
	Z	*Z* matches actual elevation data, *Z* = 147.4 meters at ground level at the threshold of runway 33.

The Unreal Editor uses a left-hand *Z*-up coordinate system, where the *Y*-axis points to the right. The aerospace aircraft blocks in Aerospace Blockset use a right-hand *Z*-down coordinate system, where the *Y*-axis points to the right. When positioning objects in a scene, keep this coordinate system difference in mind.

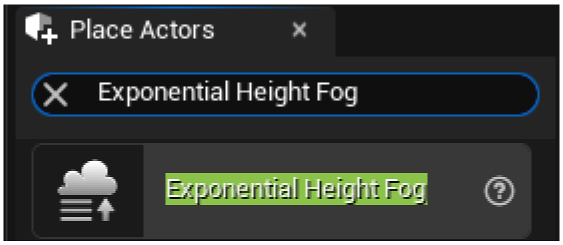
When adding assets to a scene that is in the `AutoVrtlEnv` project, you can choose from a library of aerospace-related assets. These assets are built as static meshes and begin with the prefix `SM_`. Search for these objects in the **Content Browser** pane.

For example, to add a hangar to a scene in the `AutoVrtlEnv` project:

1 In the **Content Browser** pane at the bottom of the editor, navigate to the **MathWorksAerospaceContent** folder.

2 Expand the **Environment** > **Hangar** > **Mesh** folder, or search for `SM_Hangar`. Drag the hangar from the **Content Browser** into the editing window. You can then change the position of the hangar in the editing window or on the **Details** pane on the right, in the **Transform** section.

If you want to override the default weather or use enhanced fog conditions in the scene, add the **Exponential Height Fog** actor.



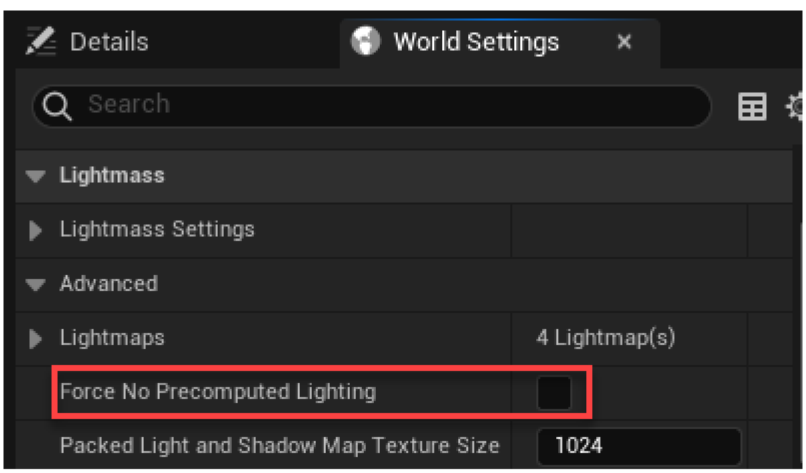
For more information on modifying scenes and adding assets, see Unreal Engine 5 Documentation.

To migrate assets from the `AutoVrtlEnv` project into your own project file, see the Unreal Engine documentation.

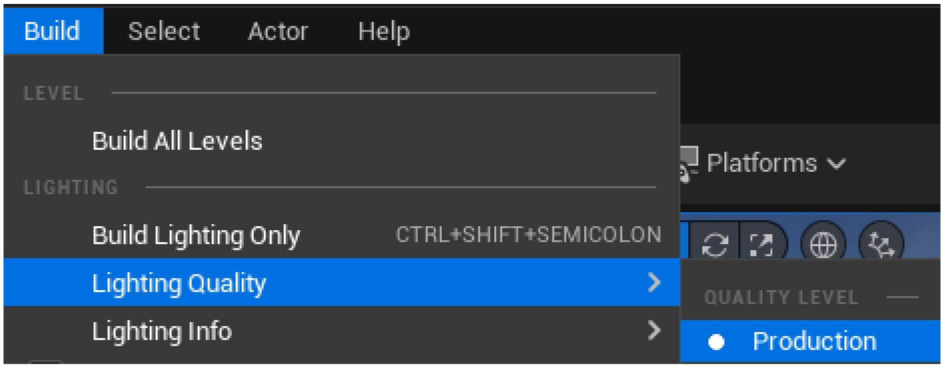
Use AutoVrtlEnv Project Lighting in Custom Scene

To use the lighting that comes installed with the `AutoVrtlEnv` project in Aerospace Blockset, follow these steps.

1 On the **World Settings** tab, clear **Force no precomputed lighting**.



2 Under **Build**, select **Lighting Quality** > **Production** to rebuild the maps with production quality. Rebuilding large maps can take time.



Run Simulation

Verify that the Simulink model and Unreal Editor are configured to co-simulate by running a test simulation.

1 In the Simulink model, click **Run**.

   Clicking **Run** starts the Simulink side of the simulation but not Unreal Editor. Continue with the next two steps to start Unreal Editor cosimulating.

2 Verify that the Diagnostic Viewer window in Simulink displays this message:

   ```
   In the Simulation 3D Scene Configuration block, you set the scene source
   to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the scene.
   ```

   This message confirms that Simulink has instantiated aircraft and other assets in the Unreal Engine 3D environment.

3 In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor. If your Simulink model contains aircraft, these aircraft can move around in the scene that is open in the editor.

To control the view of the scene during simulation, in the Simulation 3D Scene Configuration block, select the aircraft name from the **Scene view** parameter. To change the scene view as the simulation runs, first left-click inside the editor view window, then use the numeric keypad in the editor. The table shows the position of the camera displaying the scene, relative to the aircraft selected in the **Scene view** parameter.

To smoothly change the camera views, use these key commands.

Key	Camera View
1	Back left
2	Back
3	Back right
4	Left
5	Internal
6	Right
7	Front left

Key	Camera View
8	Front
9	Front right
0	Overhead

To stop a simulation, always use the stop button in Simulink and not the one in Unreal Editor.

For additional camera controls, use these key commands.

Key	Camera Control
Tab	Cycle the view between all aircraft in the scene.
Mouse scroll wheel	Control the camera distance from the aircraft.
L	Toggle a camera lag effect on or off. When you enable the lag effect, the camera view includes: • Position lag, based on the aircraft translational acceleration • Rotation lag, based on the aircraft rotational velocity This lag enables improved visualization of overall aircraft acceleration and rotation.
F	Toggle the free camera mode on or off. When you enable the free camera mode, you can use the mouse to change the pitch and yaw of the camera. This mode enables you to orbit the camera around the aircraft.

To restart a simulation, click **Run** in the Simulink model, wait until the Diagnostic Viewer displays the confirmation message, and then click **Play** in the editor. If you click **Play** before starting the simulation in your model, the connection between Simulink and the Unreal Editor is not established, and the editor displays an empty scene.

If you are co-simulating a custom project, to enable the numeric keypad, copy the `DefaultInput.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultInput.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABRelease>\...
toolbox\shared\sim3dprojects\driving\AutoVrtlEnv\Config
```

to:

```
C:\<yourproject>.project\Config
```

After tuning your custom scene based on simulation results, you can then package the scene into an executable. For more details, see "Package Custom Scenes into Executable" on page 4-15.

Reparent Actor Blueprint

Note If you are using a scene from the `AutoVtrlEnv` project that comes installed with the Aerospace Blockset Interface for Unreal Engine Projects support package, skip this section. However, if you create a new scene based off of one of the scenes in this project, then you must complete this section.

The first time that you open a custom scene from Simulink, you need to associate, or reparent, this project with the **Sim3dLevelScriptActor** level blueprint used in Aerospace Blockset. The level blueprint controls how objects interact with the 3D environment once they are placed in it. Simulink returns an error at the start of simulation if the project is not reparented. You must reparent each scene in a custom project separately.

To reparent the level blueprint, follow these steps:

1. In the Unreal Editor toolbar, select **Blueprints** > **Open Level Blueprint**.
2. In the Level Blueprint window, select **File** > **Reparent Blueprint**.
3. Click the **Sim3dLevelScriptActor** blueprint. If you do not see the **Sim3dLevelScriptActor** blueprint listed, use these steps to check that you have the `MathWorks Simulation` plugin installed and enabled:

   a. In the Unreal Editor toolbar, select **Settings** > **Plugins**.
   b. In the Plugins window, verify that the **MathWorks Interface** plugin is listed in the installed window. If the plugin is not already enabled, select the **Enabled** check box.

      If you do not see the **MathWorks Interface** plugin in this window, repeat step 3 in “Configure Environment” on page 4-4 and reopen the editor from Simulink.
   c. Close the editor and reopen it from Simulink.
4. Close the Level Blueprint window.

Install Cesium for Unreal Plugin

To customize Cesium scenes, install the Cesium for Unreal Plugin and enable the `MathWorks Geospatial` plugin.

Note The Aerospace Blockset Interface for Unreal Engine Projects supports the Cesium plugin version 1.20.1.

To download this version of the plugin, see `https://github.com/CesiumGS/cesium-unreal`.

---

1. Install the Cesium for Unreal plugin from https://cesium.com/unreal-marketplace/. Follow the directions to download the plugin.
2. Enable the `MathWorks Geospatial` plugin in the Unreal Editor. In the Unreal Editor toolbar, select **Settings** > **Plugins**.
3. In the Plugins window, verify that the **MathWorks Geospatial** plugin is listed in the installed window. If the plugin is not already enabled, select the **Enabled** check box.
4. Close the editor and reopen it from Simulink.

See Also

Simulation 3D Scene Configuration | `sim3d.Editor`

Related Examples

- Using Unreal Engine with Simulink Video Tutorial

- “Visualize with Cesium” on page 2-42

External Websites

- Unreal Engine 5 Documentation

Package Custom Scenes into Executable

When you finish modifying a custom scene as described in “Customize Scenes Using Simulink and Unreal Editor” on page 4-7, you can package the project file containing this scene into an executable. You can then configure your model to simulate from this executable by using the Simulation 3D Scene Configuration block. Executable files can improve simulation performance and do not require opening the Unreal Editor to simulate your scene. Instead, the scene runs by using the Unreal Engine that comes installed with Aerospace Blockset.

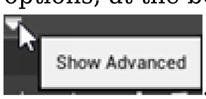
Package Scene into Executable Using Unreal Editor

Before packaging the custom scenes into an executable, make sure that the plugins are:

- Located in the Unreal Engine installation area, for example, `C:\Program Files\Epic Games\UE_5.1\Engine\Plugins\Marketplace\Mathworks`.
- Deleted from your project area, for example, `C:\project\AutoVrtlEnv\Plugins`.

Then, follow these steps.

1. Open the project containing the scene in the Unreal Editor. You must open the project from a Simulink model that is configured to co-simulate with the Unreal Editor.

   For more details on how to package projects, see "Packaging Projects" under Unreal Engine 4 Documentation.
2. In the Unreal Editor toolbar, select **Settings** > **Project Settings** to open the Project Settings window.
3. In the left pane, in the **Project** section, click **Packaging**.
4. In the **Packaging** section, set or verify the options in the table. If you do not see all these options, at the bottom of the **Packaging** section, click the **Show Advanced** expander 

.

Packaging Option	Enable or Disable
Use Pak File	Enable
Cook everything in the project content directory (ignore list of maps below)	Disable
Cook only maps (this only affects cookall)	Enable
Create compressed cooked packages	Enable
Exclude editor content while cooking	Enable

5. If you create a new scene outside the scenes provided in the support package, specify the scene from the project that you want to package into an executable. Also specify the GriffissAirport scene.

   a. In the **List of maps to include in a packaged build** option, click the **Adds Element** button .
   b. Specify the path to the scene that you want to include in the executable. By default the Unreal Editor saves maps to the `/Game/Maps` folder. For example, if the `/Game/Maps` folder

has a scene named `myScene` that you want to include in the executable, enter `/Game/Maps/myScene`. Scenes inside the plugins replace the `/Game` folder with the plugin folder.

To include the GriffissAirport map, enter `/MathWorksAerospaceContent/Maps/GriffissAirport`.

c Add or remove additional scenes as needed.

6. If you have any required asset directories to include in the executable which are not in the MathWorks plugins, then specify them under **Additional Asset Directories to Cook**.
7. In the left pane, in the **Game** section, click **Asset Manager**.
8. Expand **Primary Asset Types to Scan**, and under it expand elements **0** and **1**. For both array elements, clear the **Is Editor Only** check box
9. Close the **Project Settings** window.
10. Rebuild the lighting in your scenes. If you do not rebuild the lighting, the shadows from the light source in your executable file are incorrect and a warning about rebuilding the lighting displays during simulation. In the Unreal Editor toolbar, select **Build** > **Build Lighting Only**.
11. In the top-left menu of the editor, select **File** > **Package Project** > **Windows (64-bit)**. Select a local folder in which to save the executable, such as to the root of the project file (for example, `C:/Local/myProject`).

Note Packaging a project into an executable can take several minutes. The more scenes that you include in the executable, the longer the packaging takes.

Once packaging is complete, the folder where you saved the package contains a `WindowsNoEditor` folder that includes the executable file. This file has the same name as the project file.

Note If you repackage a project into the same folder, the new executable folder overwrites the old one.

Suppose you package a scene that is from the `myProject.uproject` file and save the executable to the `C:/Local/myProject` folder. The editor creates a file named `myProject.exe` with this path:

```
C:/Local/myProject/WindowsNoEditor/myProject.exe
```

Simulate Scene from Executable in Simulink

To improve co-simulation performance, consider configuring the Simulation 3D Scene Configuration block to co-simulate with the project executable.

1. In the Simulation 3D Scene Configuration block of your Simulink model, set the **Scene source** parameter to `Unreal Executable`.
2. Set the **File name** parameter to the name of your Unreal Editor executable file. You can either browse for the file or specify the full path to the file by using backslashes. For example:

   ```
   C:\Local\myProject\WindowsNoEditor\myProject.exe
   ```
3. Set the **Scene** parameter to the name of a scene from within the executable file. For example:

```
/MathWorksAerospaceContent/Maps/GriffissAirport
```

4 Run the simulation. The model simulates in the custom scene that you created.

If you are simulating a scene from a project that is not based on the `AutoVtrlEnv` project, then the scene simulates in full screen mode. To use the same window size as the default scenes, copy the `DefaultGameUserSettings.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultGameUserSettings.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABrelease>\...
toolbox\shared\sim3dprojects\automotive\AutoVrtlEnv\Config
```

to:

```
C:\<yourproject>.project\Config
```

Then, package scenes from the project into an executable again and retry the simulation.

See Also

Simulation 3D Scene Configuration

More About

- “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2

Get Started Communicating with the Unreal Engine Visualization Environment

You can set up communication with Unreal Engine by using the Simulation 3D Message Get and Simulation 3D Message Set blocks:

- Simulation 3D Message Get receives data from the Unreal Engine environment.
- Simulation 3D Message Set sends data to the Unreal Engine environment.

To use the blocks and communicate with Unreal Engine, make sure you install the Aerospace Blockset Interface for Unreal Engine Projects support package. For more information, see “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

Next, follow these workflow steps to set up the Simulink model and the Unreal Engine environment and run a simulation.

<table>
<tr><th colspan="2">Workflow</th><th>Description</th></tr>
<tr><td colspan="2">“Set Up Simulink Model to Send and Receive Data” on page 4-19</td><td>Configure the Simulation 3D Message Get and Simulation 3D Message Set blocks in Simulink to send and receive the cone location from Unreal Editor. The steps provides the general workflow for communicating with the editor.

The Simulation 3D Message Get and Simulation 3D Message Set blocks can send and receive these data types: `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `Boolean`. The Simulation 3D Actor Transform Set and Simulation 3D Actor Transform Get blocks can send and receive only the `single` data type.</td></tr>
<tr><td>Set Up Unreal Engine to Send and Receive Data</td><td>“C++ Workflow: Set Up Unreal Engine to Send and Receive Data” on page 4-20</td><td>Specific Unreal C++ workflow to send and receive Simulink cone location data.

• Simulation 3D Message Get receives data from an Unreal Engine environment C++ actor class. In this example workflow, you use the block to receive the cone location from Unreal Editor.
• Simulation 3D Message Set sends data to an Unreal Engine C++ actor class. In this example, you use the block to set the initial cone location in the Unreal Editor.

To follow this workflow, you should be comfortable coding with C++ in Unreal Engine. Make sure that your environment meets the minimum software requirements described in “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37.</td></tr>
</table>

Workflow		Description
	“Blueprint Workflow: Set Up Unreal Engine to Send and Receive Data” on page 4-27	Generalized Unreal Editor blueprint workflow to send and receive Simulink data.
“Run Simulation” on page 4-31		After you set up the Simulink model and Unreal Editor environment, run a simulation.

Set Up Simulink Model to Send and Receive Data

Step 1: Install Support Package

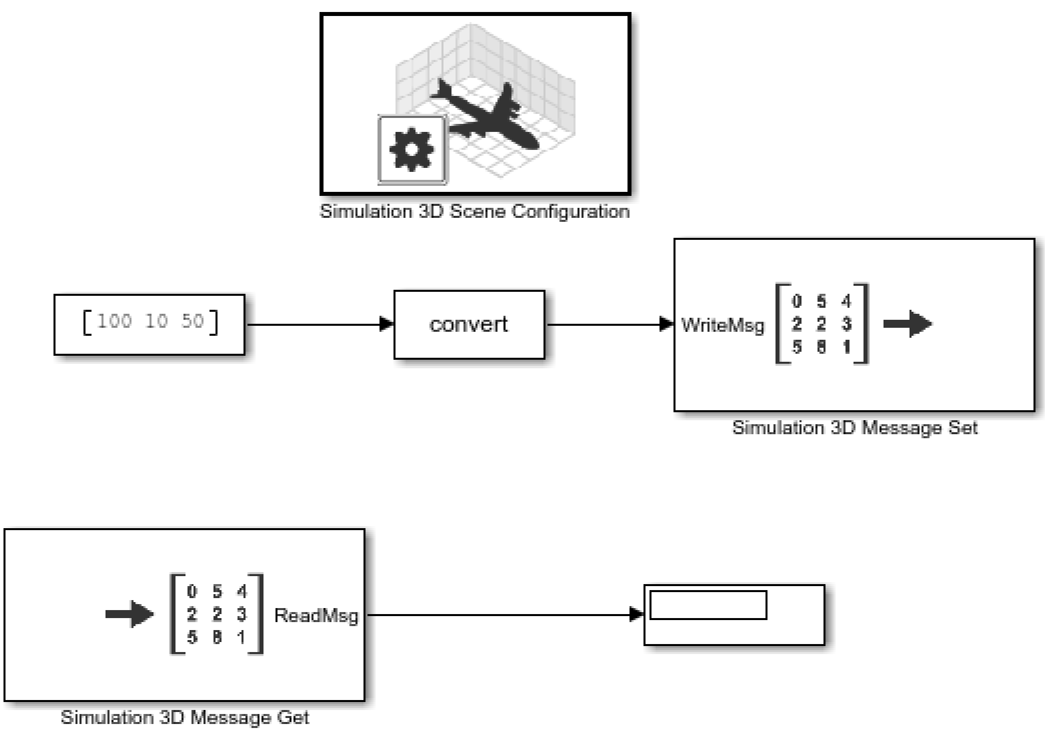
If you have already downloaded and installed Unreal Engine and the Aerospace Blockset Interface for Unreal Engine Projects support package, go to the next step.

To install and configure the support package, see “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2.

Before installing the support package, make sure that your environment meets the minimum software and hardware requirements described in “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37.

Step 2: Set Up Simulink Model

Open a new Simulink model. Connect the blocks as shown.



Step 3: Configure Blocks

Use these block settings to configure blocks to send and receive cone data from the Unreal Editor.

Block	Parameter Settings
Constant	• **Constant value** — `[100,10,50]` Sets the initial cone location in the Unreal Editor coordinate system (in cm, left-handed, in Z-up coordinate system) • **Interpret vector parameters as 1-D** — `off` • **Output data type** — `single`
Data Type Conversion	• **Output data type** — `single`
Simulation 3D Scene Configuration	• **Scene Source** — `Unreal Editor` • **Project** — *`Your_Project_Path`*`\TestSim3dGetSet.uproject` • **Open Unreal Editor** — Select to open the editor
Simulation 3D Message Get	• **Signal name, SigName** — `ConeLocGet` • **Data type, DataType** — `single` • **Message size, MsgSize** — `[1 3]` • **Sample time** — `-1`
Simulation 3D Message Set	• **Signal name, SigName** — `ConeLocSet` • **Sample time** — `-1`

C++ Workflow: Set Up Unreal Engine to Send and Receive Data

Step 4: Open Unreal Editor in Editor Mode

1. Use the Simulation 3D Scene Configuration block to open the Unreal Editor.
2. Create an Unreal Engine C++ project. Name it `TestSim3dGetSet`. For steps on how to create C ++ project, see the Unreal Engine 5 Documentation.
3. In the Unreal Editor, click the **Edit** tab in the top left corner. Select Plugins and make sure that the `MathWorks Interface` plugin is enabled. If the `MathWorks Interface` plugin is disabled, enable it and restart Unreal Editor, if prompted.
4. Close the Unreal.
5. If Visual Studio is not open, open it.
6. Add the MathWorksSimulation dependency to the `TestSim3dGetSet` project build file.

   - The project build file, `TestSim3dGetSet.Build.cs`, is located in this folder: `...\TestSim3dGetSet\Source\TestSim3dGetSet`.
   - In the build file, `TestSim3dGetSet.Build.cs`, edit the line 11 to add the "MathWorksSimulation" dependency:

   ```
    PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject",
   "Engine", "InputCore", "MathWorksSimulation"});
   ```
7. Save the change. In Visual Studio, rebuild the `TestSim3dGetSet` project. Close Visual Studio.

Tip Before rebuilding the project in Visual Studio, make sure that Unreal is not open.

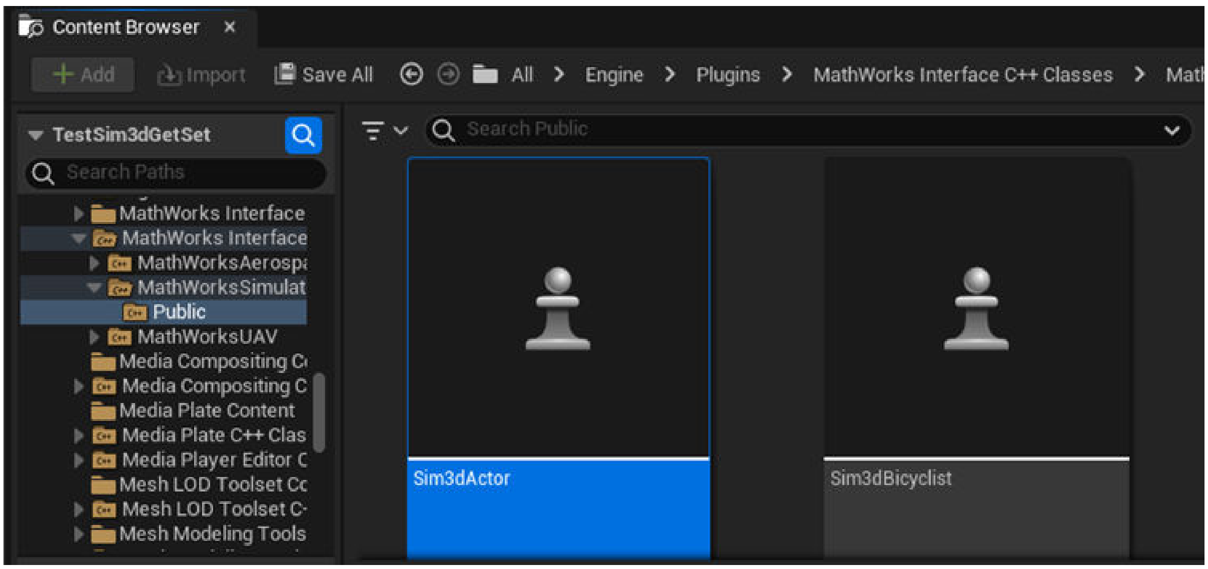
8 Start MATLAB. Change the current folder to the location of the Unreal Engine `TestSim3dGetSet` project.

9 In MATLAB, open the project:

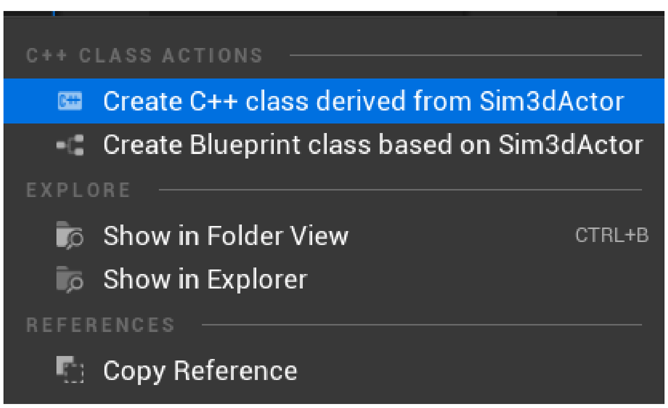
```
editor = sim3d.Editor('TestSim3dGetSet.uproject');
editor.open();
```

Step 5: Create Actor Class

1 In the Unreal Editor, from the MathWorksSimulation C++ classes directory, select **Sim3dActor**.



Right-click and select **Create C++ class derived from Sim3dActor**.



2 Name the new Sim3dActor `SetGetActorLocation`. Select **Public**. Click **Create Class**.

3 Close the Unreal Editor.

Step 6: Open SetGetActorLocation.h

Visual Studio opens with new C++ files in the project folder:

- `SetGetActorLocation.h`
- `SetGetActorLocation.cpp`

Make sure you close the Unreal Editor.

In Visual Studio, build the solution `TestSim3dGetSet`:

1. In the Solution Explorer, right-click **Solution 'TestSim3dGetSet' (2 projects)**.
2. Select **Build Solution**.
3. After the solution builds, open `SetGetActorLocation.h`. Edit the file as shown.

Replacement Code: `SetGetActorLocation.h`

This is the replacement code for `SetGetActorLocation.h`.

```
// Copyright 2019 The MathWorks, Inc.

#pragma once

#include "Sim3dActor.h"
#include "SetGetActorLocation.generated.h"

UCLASS()
class TESTSIM3DGETSET_API ASetGetActorLocation : public ASim3dActor
{
    GENERATED_BODY()

    void *SignalReader;
    void *SignalWriter;

public:
    // Sets default values for this actor's properties
    ASetGetActorLocation();

    virtual void Sim3dSetup() override;
    virtual void Sim3dRelease() override;
    virtual void Sim3dStep(float DeltaSeconds) override;
};
```

Step 7: Open SetGetActorLocation.cpp

Open `SetGetActorLocation.cpp` and replace the block of code.

Replacement Code: Set Pointer to Parameter

This code allows you to set a pointer to the parameter Signal Name parameter for the Simulink blocks Simulation 3D Message Set and Simulation 3D Message Get, respectively.

```
// Sets default values
ASetGetActorLocation::ASetGetActorLocation():SignalReader(nullptr), SignalWriter(nullptr)
{
}
```

Replacement Code: Access Actor Tag Name

The following code allows you to access the tag name of this actor after it is instantiated in the scene with an assigned tag name. The code also initializes the pointers `SignalReader` and `SignalWriter`, to initiate a link between Unreal Editor and Simulink. The variables represent these block Signal Name parameter values:

- `SignalReaderTag` — Simulation 3D Message Set

- `SignalWriterTag` — Simulation 3D Message Get

```
void ASetGetActorLocation::Sim3dSetup()
{
Super::Sim3dSetup();
	if (Tags.Num() != 0) {
		unsigned int numElements = 3;
		FString tagName = Tags.Top().ToString();

		FString SignalReaderTag = tagName;
		SignalReaderTag.Append(TEXT("Set"));
		SignalReader = StartSimulation3DMessageReader(TCHAR_TO_ANSI(*SignalReaderTag), sizeof(float)*numElements);

		FString SignalWriterTag = tagName;
		SignalWriterTag.Append(TEXT("Get"));
		SignalWriter = StartSimulation3DMessageWriter(TCHAR_TO_ANSI(*SignalWriterTag), sizeof(float)*numElements);
		}
}
```

Additional Code: Read and Write Data During Run Time

Add this code to allow Unreal Engine to read the data value set by Simulation 3D Message Set and then write back to Simulation 3D Message Get during run time. Unreal Engine uses this data to set the location value of the actor.

```
void ASetGetActorLocation::Sim3dStep(float DeltaSeconds)
{
	unsigned int numElements = 3;
	float array[3];
	int statusR = ReadSimulation3DMessage(SignalReader, sizeof(float)*numElements, array);
	FVector NewLocation;
	NewLocation.X = array[0];
	NewLocation.Y = array[1];
	NewLocation.Z = array[2];
	SetActorLocation(NewLocation);
	float fvector[3] = { NewLocation.X, NewLocation.Y, NewLocation.Z };
	int statusW = WriteSimulation3DMessage(SignalWriter, sizeof(float)*numElements ,fvector);
}
```

Additional Code: Stop Simulation

Add this code so that Unreal Engine stops when you press the Simulink stop button. The code destroys the pointer SignalReader and SignalWriter.

```
void ASetGetActorLocation::Sim3dRelease()
{
	Super::Sim3dRelease();
	if (SignalReader) {
		StopSimulation3DMessageReader(SignalReader);
	}
	SignalReader = nullptr;

	if (SignalWriter) {
		StopSimulation3DMessageWriter(SignalWriter);
	}
	SignalWriter = nullptr;
}
```

Entire Replacement Code: `SetGetActorLocation.cpp`

This is the entire replacement code for `SetGetActorLocation.cpp`.

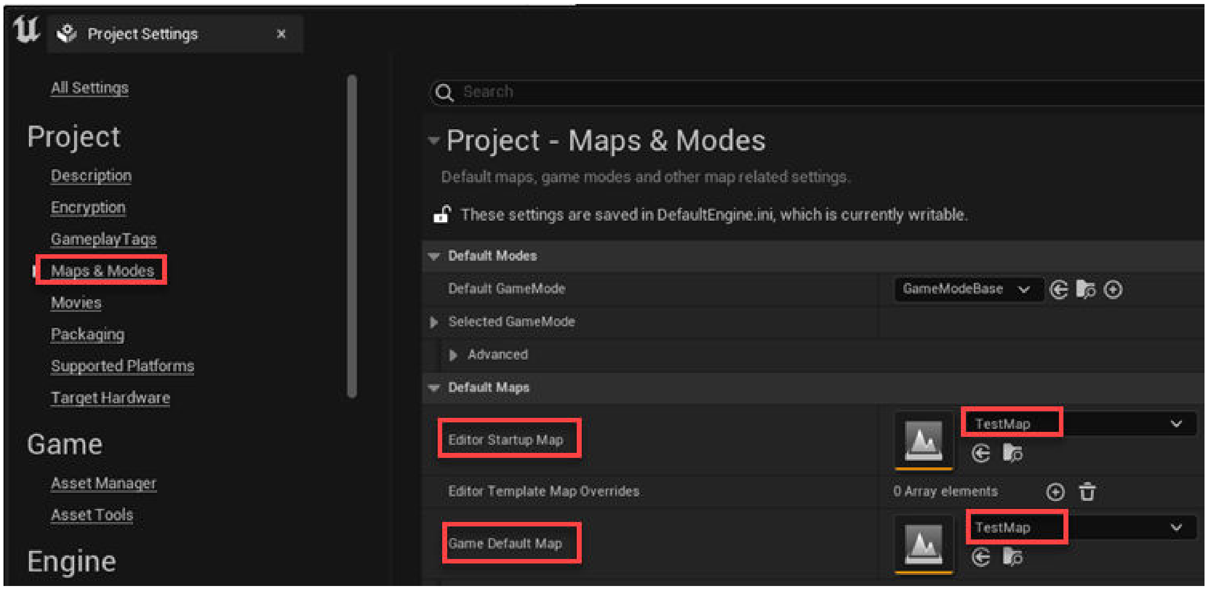
```
// Copyright 2019 The MathWorks, Inc.
#include "SetGetActorLocation.h"

// Sets default values
ASetGetActorLocation::ASetGetActorLocation():SignalReader(nullptr), SignalWriter(nullptr)
```

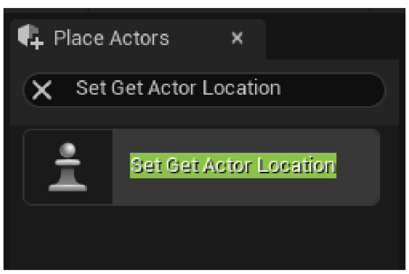
```
{
}

void ASetGetActorLocation::Sim3dSetup()
{
Super::Sim3dSetup();
       if (Tags.Num() != 0) {
              unsigned int numElements = 3;
              FString tagName = Tags.Top().ToString();

              FString SignalReaderTag = tagName;
              SignalReaderTag.Append(TEXT("Set"));
              SignalReader = StartSimulation3DMessageReader(TCHAR_TO_ANSI(*SignalReaderTag), sizeof(float)*numElements);

              FString SignalWriterTag = tagName;
              SignalWriterTag.Append(TEXT("Get"));
              SignalWriter = StartSimulation3DMessageWriter(TCHAR_TO_ANSI(*SignalWriterTag), sizeof(float)*numElements);
              }
}

void ASetGetActorLocation::Sim3dStep(float DeltaSeconds)
{
       unsigned int numElements = 3;
       float array[3];
       int statusR = ReadSimulation3DMessage(SignalReader, sizeof(float)*numElements, array);
       FVector NewLocation;
       NewLocation.X = array[0];
       NewLocation.Y = array[1];
       NewLocation.Z = array[2];
       SetActorLocation(NewLocation);
       float fvector[3] = { NewLocation.X, NewLocation.Y, NewLocation.Z };
       int statusW = WriteSimulation3DMessage(SignalWriter, sizeof(float)*numElements ,fvector);
}

void ASetGetActorLocation::Sim3dRelease()
{
       Super::Sim3dRelease();
       if (SignalReader) {
              StopSimulation3DMessageReader(SignalReader);
       }
       SignalReader = nullptr;

       if (SignalWriter) {
              StopSimulation3DMessageWriter(SignalWriter);
       }
       SignalWriter = nullptr;
}
```

Step 8: Build the Visual Studio Project and Open Unreal Editor Using the Block

Press **F5** on the keyboard to run the Visual Studio solution `TestSim3dGetSet`. The Unreal Editor opens.

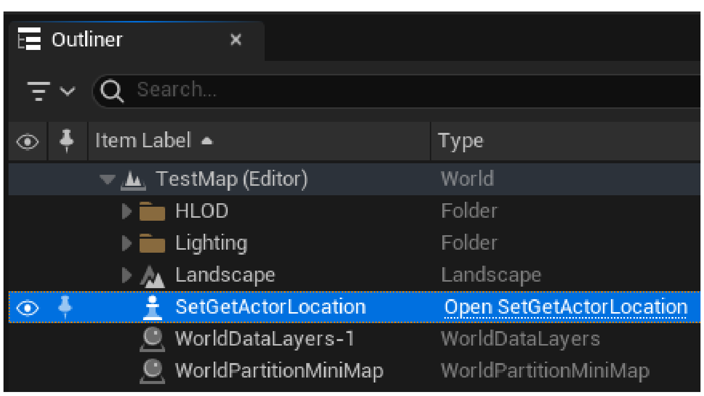
Note In the Unreal Editor, save the current level by clicking **Save Current** (located in the top left) and name it `TestMap`. Add this level as default to Project Settings by clicking on **Edit** > **Project Settings** > **Maps&Modes**. Then select `TestMap` as the default value for the Editor Startup Map and Game Default Map. Close Project Settings to save the default values.





Step 9: Check Actor

On the **World Outliner** tab, check that the new instantiated actor, `SetGetActorLocation1`, is listed.

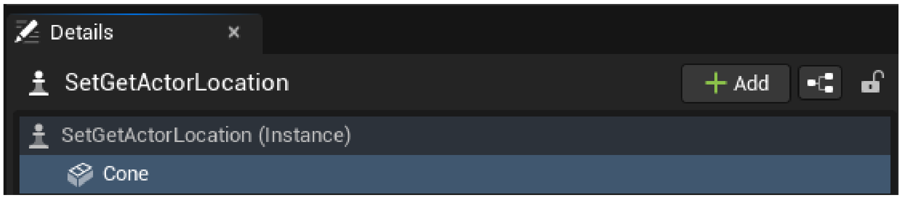


Step 10: Add Mesh

Click on the actor that you created in “Step 9: Check Actor” on page 4-25.

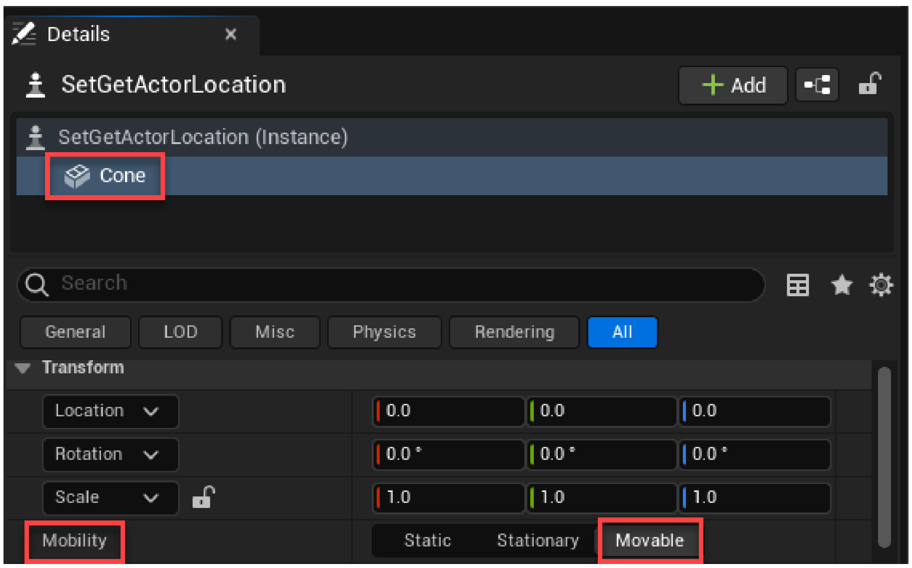
1 In the **Details** panel, click on Add Component to add a mesh to the actor `SetConeLocation1`. Choose `Cone` as the default mesh.

2 Find the property tags for actor `SetConeLocation1`. Add a tag by clicking on the plus sign next to 0 Array elements. Name it `ConeLoc`.



Step 11: Set Cone Location

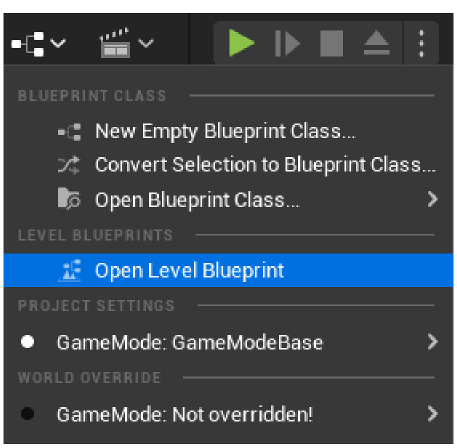
On the **Details** tab, click **Cone**. Set the cone to X = `0.0`, Y = `0.0`, and Z = `0.0`. Also set the actor **Mobility** property to Movable.



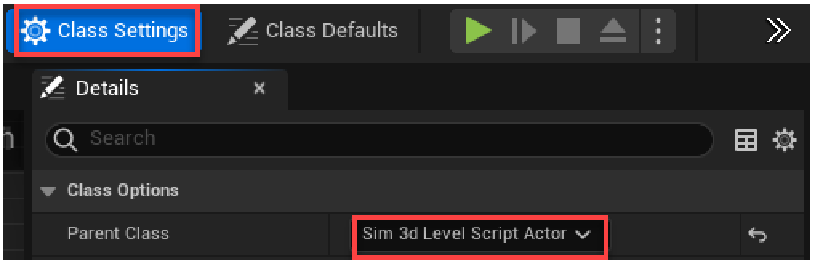
Step 12: Set Parent Class and Save Scene

Set the parent class.

1 Under **Blueprints**, click **Open Level Blueprint**, and select **Class Settings**.



2 In the **Class Options**, set **Parent Class** to `Sim 3d Level Script Actor`.



Save the Unreal Editor scene.

Step 13: Run Simulation

Run the simulation. Go to "Run Simulation" on page 4-31.

Reference: C++ Functions for Sending and Receiving Simulink Data

Call these C++ functions from `Sim3dSetup`, `Sim3dStep`, and `Sim3dRelease` to send and receive Simulink data.

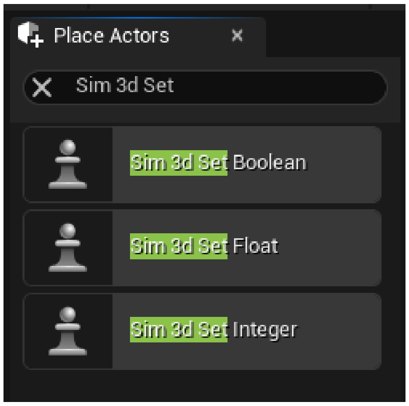
To	C++ Functions
Receive data	`StartSimulation3DMessageReader`
	`ReadSimulation3DMessage`
	`StopSimulation3DMessageReader`
Send data	`StartSimulation3DMessageWriter`
	`WriteSimulation3DMessage`
	`StopSimulation3DMessageWriter`

Blueprint Workflow: Set Up Unreal Engine to Send and Receive Data

Step 4: Configure Scenes to Receive Data

To use the Simulation 3D Message Set block, you must configure scenes in the Unreal Engine environment to receive data from the Simulink model:

1 In the Unreal Editor, instantiate the `Sim3DGet` actor that corresponds to the data type you want to receive from the Simulink model. This example shows the Unreal Editor `Sim3DGet` data types.

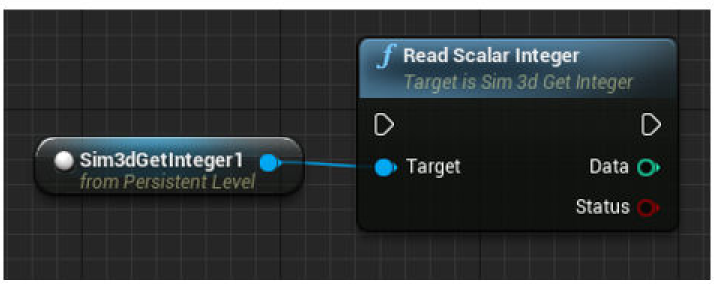


2 Specify an actor tag name that matches the Simulation 3D Message Set block **Signal name** parameter.

3 Navigate to the Level Blueprint.

4 Find the blueprint method for the `Sim3DGet` actor class based on the data type and size that you want to receive from the Simulink model.

   For example, in Unreal Editor, this diagram shows that `Read Scalar Integer` is the method for `Sim3DGetInteger` actor class to receive `int32` data type of size scalar.

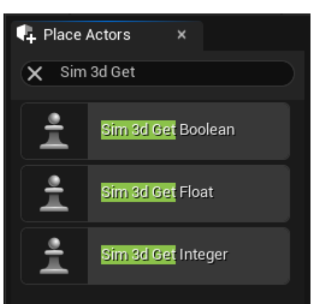


5 Compile and save the scene.

Step 5: Configure Scenes to Send Data

To configure scenes in the Unreal Engine environment to send data to the Simulink model:

1 In the Unreal Editor, instantiate the `Sim3DSet` actor that corresponds to the data type you want to send to the Simulink model. This example shows the Unreal Editor `Sim3DSet` data types.



2 Specify an actor tag name that matches the Simulation 3D Message Get block **Signal name** parameter.

3 Navigate to the Level Blueprint.

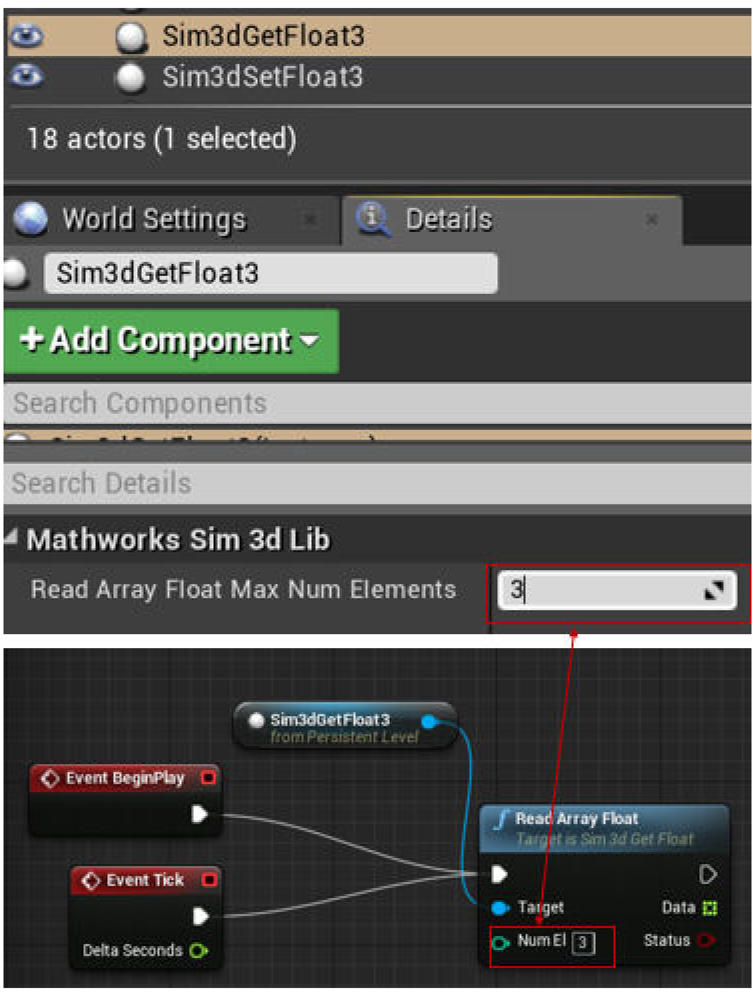
4 Find the blueprint method for the `Sim3DSet` actor class based on the data type and size specified by the Simulation 3D Message Get block **Data type** and **Message size** parameters.

   For this example, the array size is 3. The Unreal Editor diagram shows that `Write Array Float` is the method for the `Sim3DSetFloat3` actor class that sends float data type of array size 3.

5 Compile and save the scene.

Note Optionally, for better performance, set **Read Array Float Max Num Elements** to `Num El` in the Actor Blueprint.



Step 6: Create Blueprint

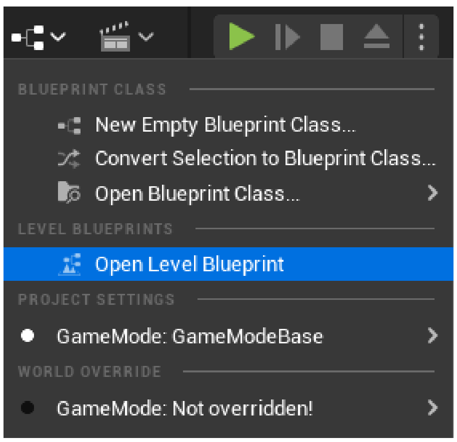
In the Unreal Editor, create a level blueprint connecting the Get and Set actors.

1 Set the actor tag values.

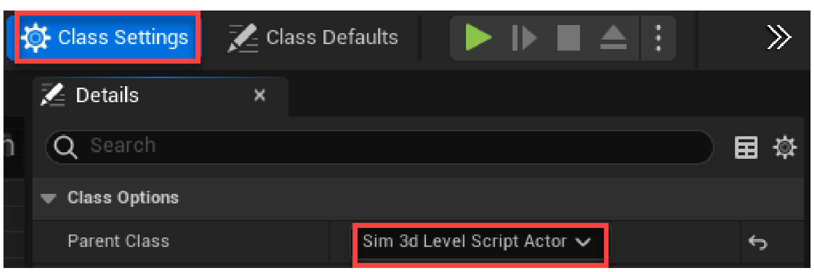
- `Sim3dGetFloat1` — Simulation 3D Message Set block **Signal name, SigName** parameter value, for example `ConeLocSet`
- `Sim3dSetFloat1` — Simulation 3D Message Get block **Signal name, SigName** parameter value, for example `ConeLocGet`

2 Set the parent class.

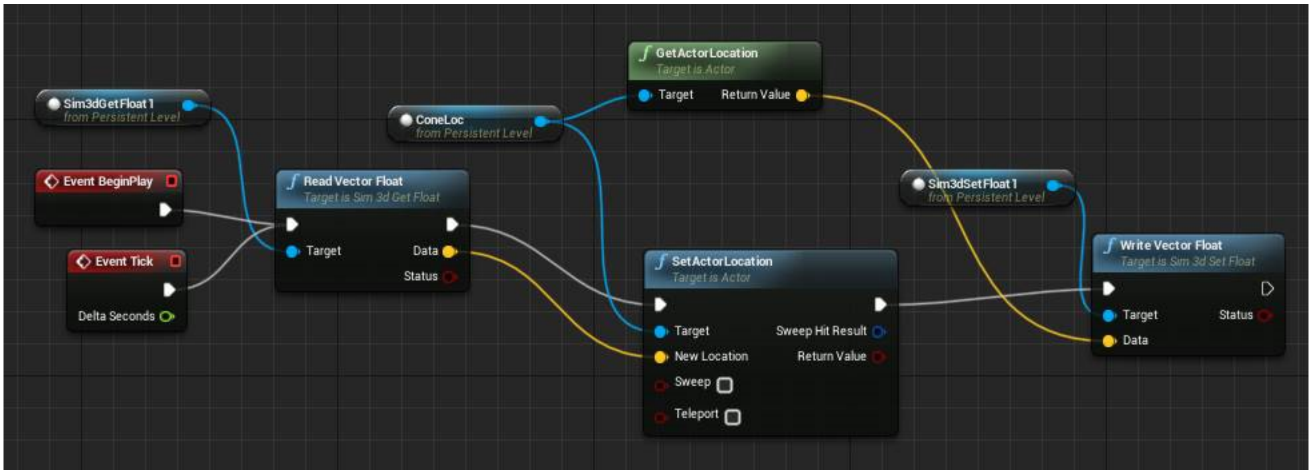
a Under **Blueprints**, click **Open Level Blueprint**, and select **Class Settings**.



b In the **Class Options**, set **Parent Class** to `Sim 3d Level Script Actor`.



3 In the level blueprint, make the connections, for example:



Step 7: Run Simulation

Run the simulation. Go to "Run Simulation" on page 4-31.

Run Simulation

After you configure the Simulink model and Unreal Editor environment, you can run the simulation.

Note At the BeginPlay event, Simulink does not receive data from the Unreal Editor. Simulink receives data at Tick events.

Run the simulation.

1. In the Simulink model, click **Run**.

   Because the source of the scenes is the project opened in the Unreal Editor, the simulation does not start.
2. Verify that the Diagnostic Viewer window in Simulink displays this message:

   ```
   In the Simulation 3D Scene Configuration block, you set the scene source
   to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the scene.
   ```

   This message confirms that Simulink has instantiated the vehicles and other assets in the Unreal Engine 3D environment.
3. In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor.

You can send and receive these data types: `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`. The code in "Step 7: Open SetGetActorLocation.cpp" on page 4-22 reads single data type values (or float values) from Simulink.

See Also

`ASim3dActor` | `Sim3dSetup` | `Sim3dStep` | `Sim3dRelease` | Simulation 3D Scene Configuration | Simulation 3D Message Get | Simulation 3D Message Set

External Websites

- Unreal Engine
- Unreal Engine 5 Documentation

Griffiss Airport Lighting

To help you visualize your flightpath workflows, the Aerospace Blockset Interface for Unreal Engine Projects support package provides a map of the Griffiss International Airport in Rome, New York. The airport map includes static mesh models of every taxiway and runway light, Medium Intensity Approach Light System with Runway Alignment Indicator Lights (MALSR), and building lights.

Turn Airport Lights On and Off

To turn on and off the lights for the runway, taxiway, threshold, and MALSR, use the **Lights On** switch.

1 Load the Griffiss Airport map in the Unreal Editor by double-clicking `GriffissAirport` in the `Maps` folder.

2 At the top center of the view, click **Blueprints** > **Open Level Blueprint**.

   The **Griffiss Airport** > **Event Graph** tab of the level blueprint appears.

3 In the **Details** panel on the right, see the **Defaults** section.

   Tip If the **Details** panel is not visible, enable it by clicking **Windows** > **Details** in the top left menu.

4 To turn on lights, select the **Details** > **Defaults** > **Lights On** check box.

   To turn off lights, clear the **Details** > **Defaults** > **Lights On** check box.

5 Save your changes and exit the level blueprint.

6 Run the simulation.

Niagara lights are created in the level blueprint at the start of the simulation. They are not visible if the simulation is not running.

See Also

Simulation 3D Scene Configuration

More About

- “How 3D Simulation for Aerospace Blockset Works” on page 2-40
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37

External Websites

- Unreal Engine
- Unreal Engine 5 Documentation

Prepare Custom Aircraft Mesh for the Unreal Editor

This example shows you how to create an aircraft mesh that is compatible with the project in the Aerospace Blockset Interface for Unreal Engine Projects support package. For illustration purposes, it uses Blender®.You can specify the mesh in the Simulation 3D Aircraft block to visualize the aircraft in the Unreal Editor when you run a simulation.

Before you start, install the Aerospace Blockset Interface for Unreal Engine Projects support package. See "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2.

To create a compatible custom aircraft mesh, follow these workflow steps.

Step	Description
"Step 1: Check Units and Axes" on page 4-34	In a 3-D creation environment, set up the units and axes for the mesh.
"Step 2: Set Up Bone Hierarchy" on page 4-35	Set up the aircraft mesh bone hierarchy and specify part names.
"Step 3: Connect Mesh to Skeleton" on page 4-37	Parent the entire mesh to the armature.
"Step 4: Assign Materials" on page 4-37	Optionally, assign materials to the aircraft parts.
"Step 5: Export Mesh and Armature" on page 4-37	Export the aircraft mesh and armature in `.fbx` file format.
"Step 6: Import Mesh to Unreal Editor" on page 4-37	Import the aircraft mesh into the Unreal Editor.
"Step 7: Set Block Parameters" on page 4-39	Set up the block parameters.

Note To create the mesh, this example uses the 3-D creation software Blender Version 2.93.

Step 1: Check Units and Axes

1. Create or import an aircraft mesh into a 3-D modeling tool, for example, Blender.
2. Check that the **Length** unit is set to `Centimeters` and the **Rotation** unit is set to `Degrees`.

   If the units are not correctly set, then correct them. For example, in Blender, use steps like these:

   a. Change **Unit** scale from `1.0` to `0.01`, and **Length** from `Meters` to `Centimeters`.
   b. Check the dimensions of some mesh objects.

      Units should be centimeters and the sizes should be 100 times too small.
   c. Select the entire mesh and scale it by 100 in all three dimensions.
   d. Position the mesh so that the global axes origin is near the center of mass, with *X* pointing forward and *Z* pointing upward.

e To complete the transformation, click **Object** > **Apply** > **Location** and **Object** > **Apply** > **Scale**.

Step 2: Set Up Bone Hierarchy

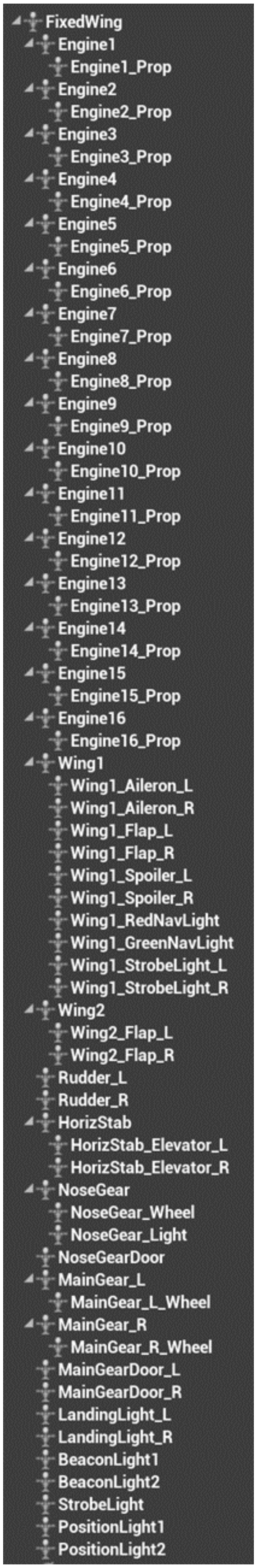
1. Create an armature, if necessary, and use the following naming convention for the bones to ensure compatibility with the animation components in the support package. Make sure to follow the bone hierarchy shown.

   Note You can omit or add bones and still maintain compatibility with the custom aircraft skeleton in the support package, as long as the rules for sharing skeleton assets are met.

2. Most bones share the vertical, global-*z*-axis-aligned orientation of the root bone.

   - Align wheel bones with the global *y*-axis.
   - Align a control surface (such as an aileron) bone perpendicular to its surface and rotate it to align with the surface hinge line.

3. Check that no mesh elements have the same names as any of the bones. Rename them as necessary.


FixedWing
Engine1
Engine1_Prop
Engine2
Engine2_Prop
Engine3
Engine3_Prop
Engine4
Engine4_Prop
Engine5
Engine5_Prop
Engine6
Engine6_Prop
Engine7
Engine7_Prop
Engine8
Engine8_Prop
Engine9
Engine9_Prop
Engine10
Engine10_Prop
Engine11
Engine11_Prop
Engine12
Engine12_Prop
Engine13
Engine13_Prop
Engine14
Engine14_Prop
Engine15
Engine15_Prop
Engine16
Engine16_Prop
Wing1
Wing1_Aileron_L
Wing1_Aileron_R
Wing1_Flap_L
Wing1_Flap_R
Wing1_Spoiler_L
Wing1_Spoiler_R
Wing1_RedNavLight
Wing1_GreenNavLight
Wing1_StrobeLight_L
Wing1_StrobeLight_R
Wing2
Wing2_Flap_L
Wing2_Flap_R
Rudder_L
Rudder_R
HorizStab
HorizStab_Elevator_L
HorizStab_Elevator_R
NoseGear
NoseGear_Wheel
NoseGear_Light
NoseGearDoor
MainGear_L
MainGear_L_Wheel
MainGear_R
MainGear_R_Wheel
MainGearDoor_L
MainGearDoor_R
LandingLight_L
LandingLight_R
BeaconLight1
BeaconLight2
StrobeLight
PositionLight1
PositionLight2

Step 3: Connect Mesh to Skeleton

1 Parent the entire mesh to the armature, for example, Blender, in **Object Mode**:

   a Select the entire mesh.

   b Click **Shift+Left** on one of the bones in the viewport.

   c To display the parenting menu, press **Ctrl+P**, and choose **Armature Deform with Empty Groups** to create an empty mesh **Vertex Group** for every bone.

2 For each mesh object:

   a Assign weight to the appropriate **Vertex Group**.

   b Add an `Armature` modifier for that **Vertex Group**.

Step 4: Assign Materials

Optionally, assign material slots to the aircraft parts. The first material slot should correspond to the aircraft body. The Simulation 3D Aircraft block sets only the first slot material (i.e. color) assignment.

Step 5: Export Mesh and Armature

Export the mesh and armature in the `.fbx` file format, for example, in Blender:

1 On the **Object Types** pane, select **Armature** and **Mesh**.

2 On the **Transform** pane, set:

   - **Scale** to 1.00
   - **Apply Scalings** to `All Local`
   - **Forward** to `X Forward`
   - **Up** to `Z Up`

   Select **Apply Unit**.

   Select **Use Space Transform**.

3 On the **Geometry** pane:

   - Set **Smoothing** to `Face`.
   - Select **Apply Modifiers**.

4 On the **Armature** pane, set:

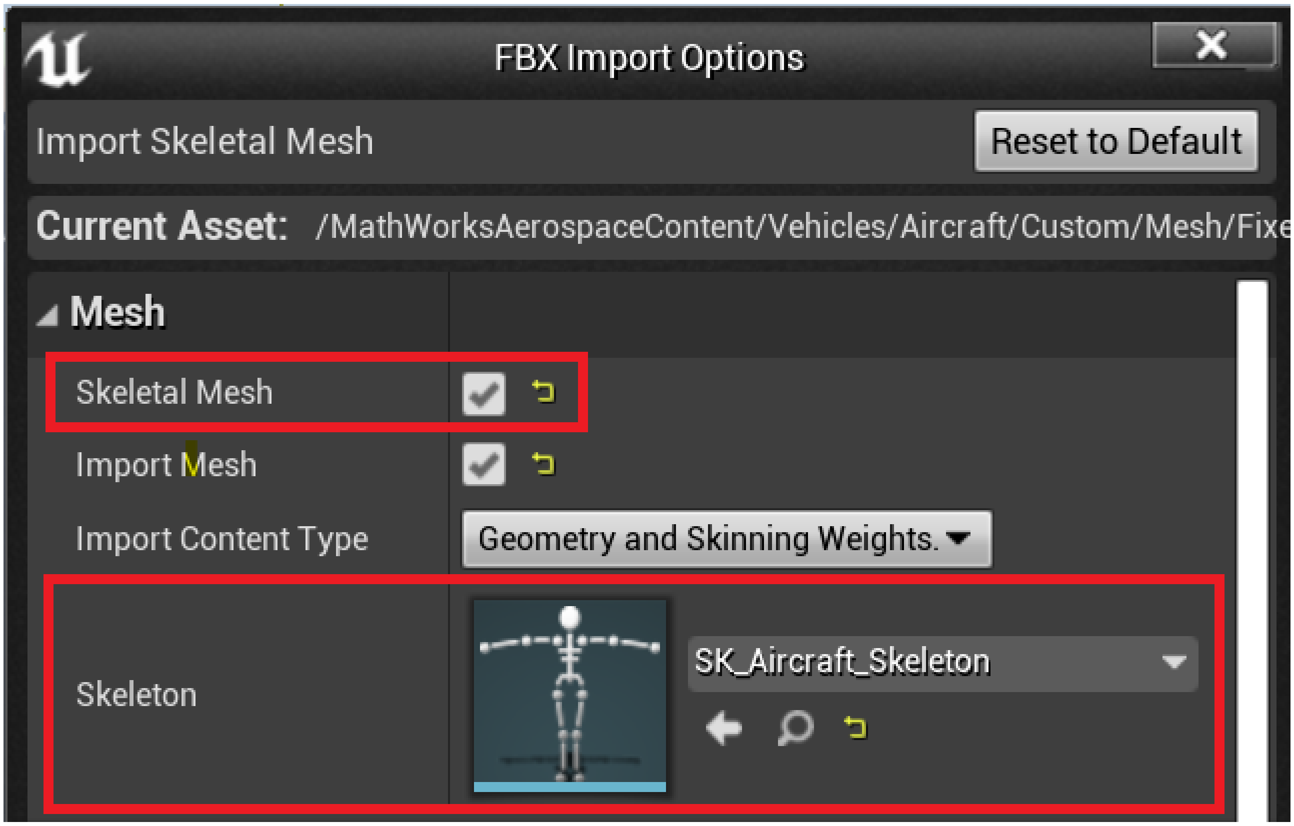
   - **Primary Bone Axis** to `Z Axis`.
   - **Secondary Bone Axis** to `X Axis`.
   - Armature **FBXNode Type** to `Null`.

5 Clear **Bake Animation**, then select **Export FBX**.

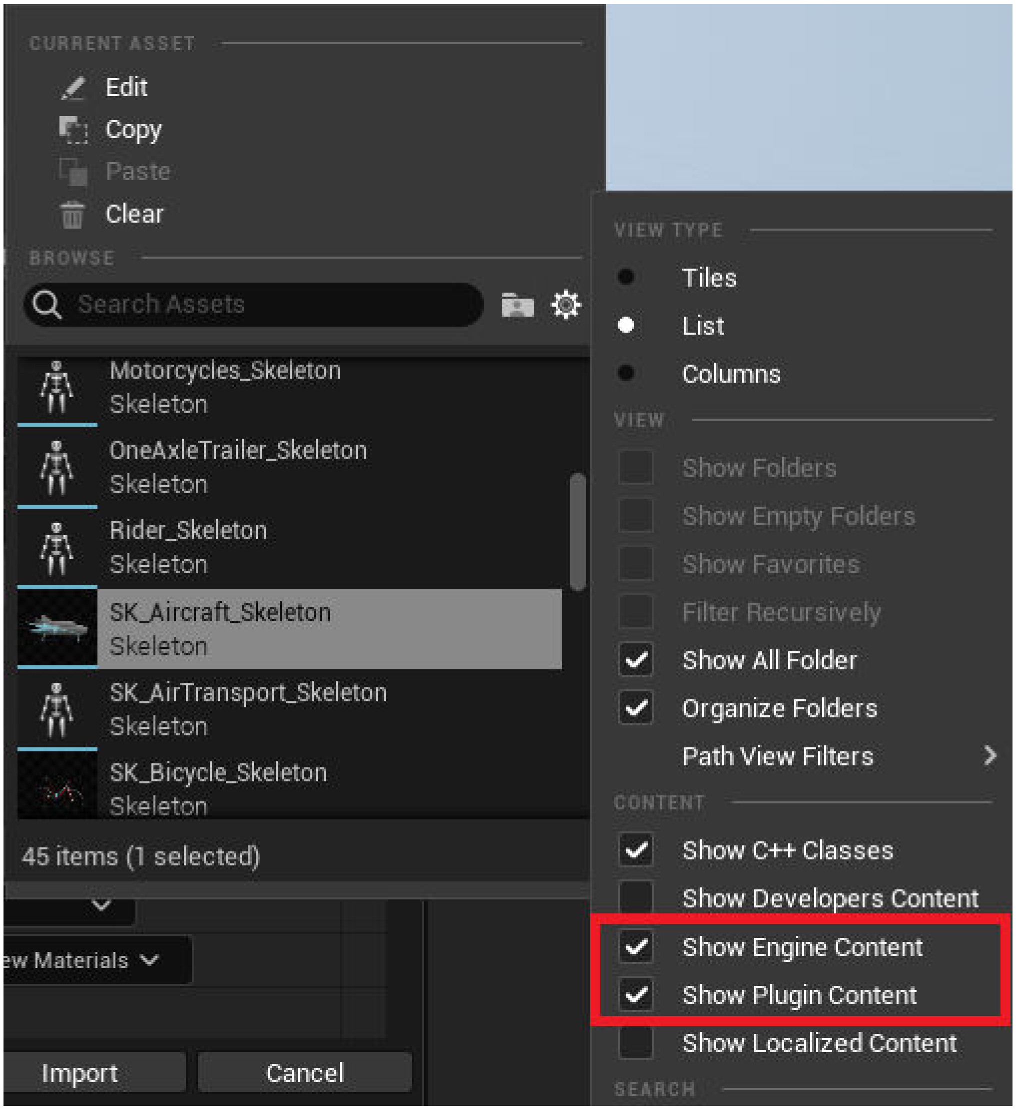
Step 6: Import Mesh to Unreal Editor

1 Open the Unreal Engine `AutoVrtlEnv.uproject` project in the Unreal Engine Editor.

2 In the editor, import the FBX® file as a skeletal mesh. Assign the **Skeleton** to the `SK_Aircraft_Skeleton` asset.



Note If you cannot select `SK_Aircraft_Skeleton`, click the gearbox. Make sure **Show Engine Content** and **Show Plugin Content** are both selected.



3 Open the imported mesh and assign materials to each material slot.

Step 7: Set Block Parameters

In your Simulink model, set these Simulation 3D Aircraft block parameters:

- **Type** to `Custom`.
- **Path** to the path in the Unreal Engine project that contains the imported mesh. For example, if a mesh named SK_X15 is imported into the `Vehicles/Aircraft/Custom/Mesh` folder, then the full path is `/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_X15.SK_X15`.

See Also

Simulation 3D Scene Configuration | Simulation 3D Aircraft

More About

- “How 3D Simulation for Aerospace Blockset Works” on page 2-40
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37

External Websites

- Blender

Place Cameras on Actors in the Unreal Editor

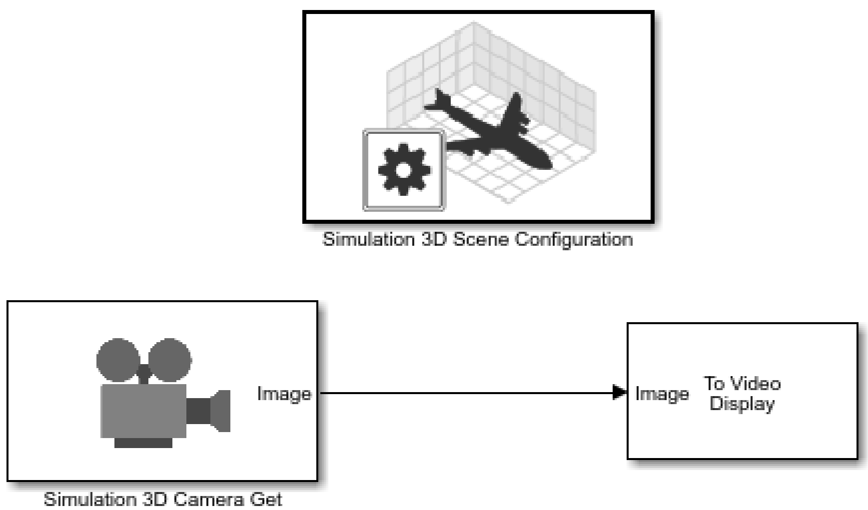
To visualize objects in an Unreal Editor scene, you can place cameras on static or custom actors in the scene. To start, you need the Aerospace Blockset Interface for Unreal Engine Projects support package. See “Install Support Package and Configure Environment” on page 4-3.

To follow this workflow, you should be comfortable using Unreal Engine. Make sure that you have Visual Studio 2022 installed on your computer.

Place Camera on Static Actor

Follow these steps to place a Simulation 3D Camera Get block that is offset from a cone in the Unreal Editor. Although this example uses the To Video Display block from Computer Vision Toolbox, you can use a different visualization block to display the image.

1 In a Simulink model, add the Simulation 3D Scene Configuration, Simulation 3D Camera Get, and To Video Display blocks.

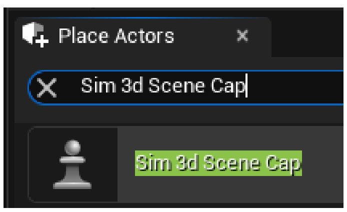


Set these block parameters. In the Simulation 3D Scene Configuration block, select **Open Unreal Editor**.

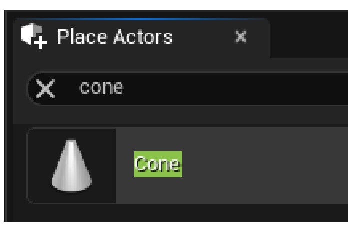
Block	Parameter Settings
Simulation 3D Scene Configuration	• **Scene Source** — `Unreal Editor` • **Project** — Specify the path and name of the support package project file. For example, `C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject`

Block	Parameter Settings
Simulation 3D Camera Get	• **Sensor identification** — 1 • **Vehicle name** — `Scene Origin` • **Vehicle mounting location** — `Origin` • **Specify offset** — `on` • **Relative translation [X, Y, Z]** — `[-6, 0, 2]` This offsets the camera location from the cone mounting location, 6 m behind, and 2 m up. • **Relative rotation [Roll, Pitch, Yaw]** — `[0, 15, 0]`

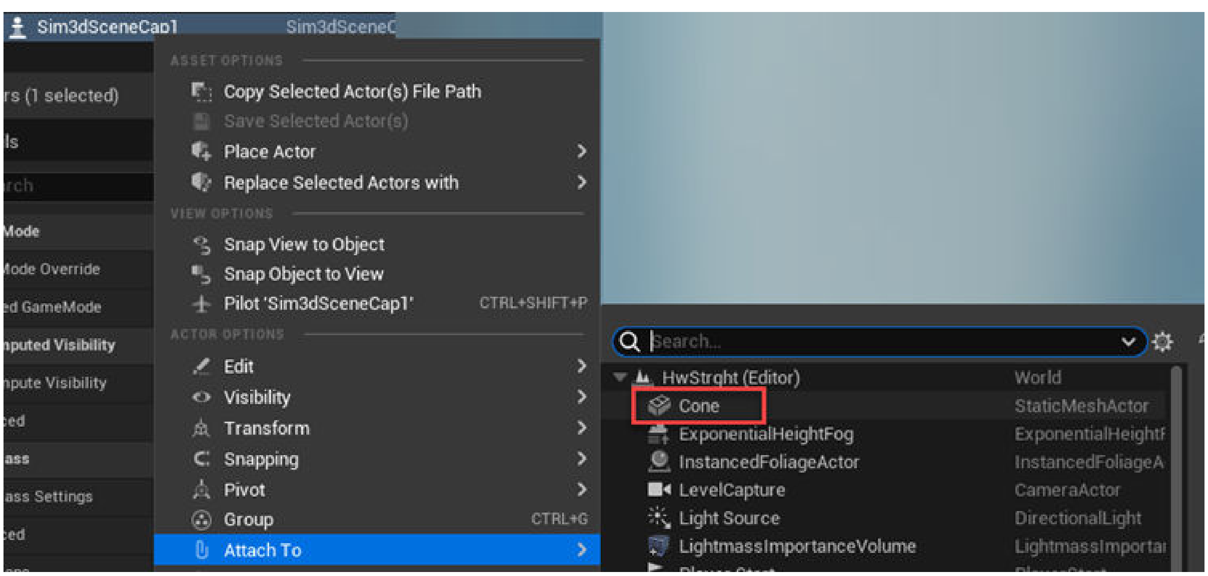
2 In the Unreal Editor, from the **Place Actors** tab, add a **Sim 3d Scene Cap** to the world, scene, or map.



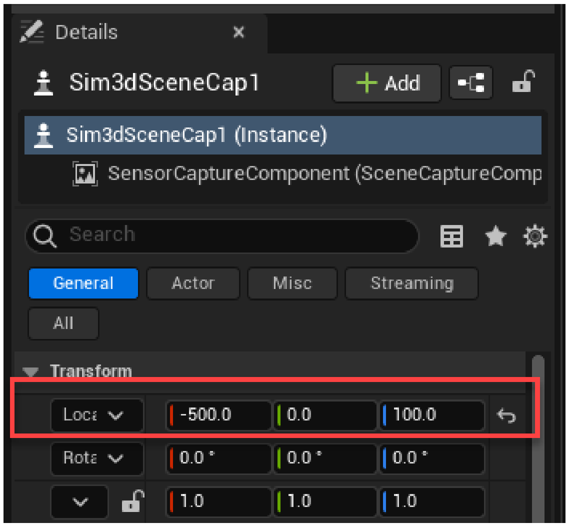
3 In the Unreal Editor, from the **Place Actors** tab, add a **Cone** to the world, scene, or map.



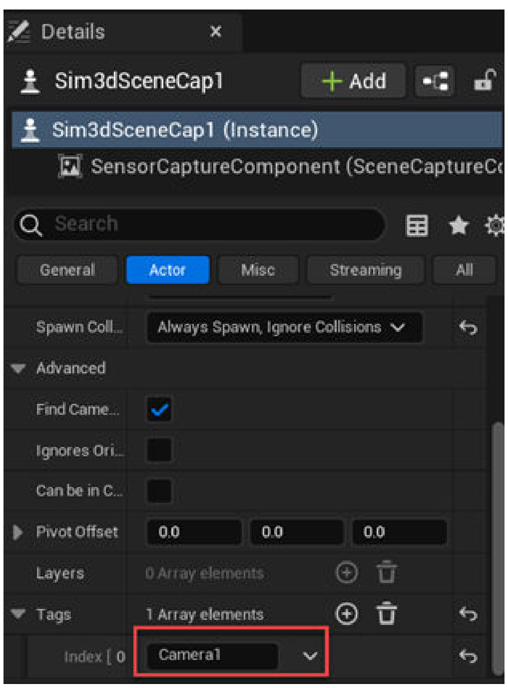
4 On the **World Outliner tab**, right-click the **Sim3DSceneCap1** and attach it to the **Cone**.



5 On the **Details** tab, under **Transform**, add a location offset of `-500,0,100` in the X, Y, and Z world coordinate system, respectively. This attaches the camera 500 cm behind the cone and 100 cm above it. The values match the Simulation 3D Camera Get block parameter **Relative translation [X, Y, Z]** value.



6 On the **Details** tab, under **Actor**, tag the **Sim3DSceneCap1** with the name Camera1.



7 Run the simulation.

a In the Simulink model, click **Run**.

Because the source of the scenes is the project opened in the Unreal Editor, the simulation does not start.

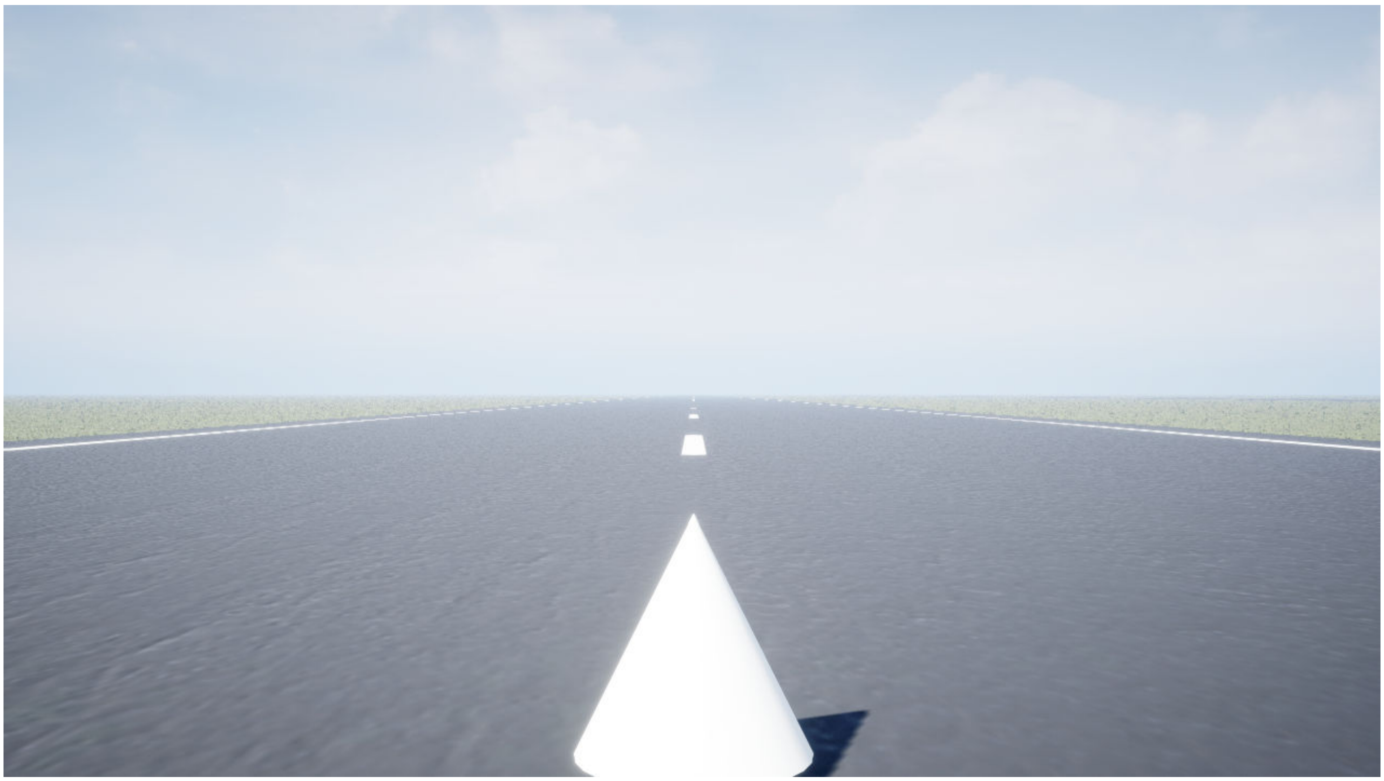
b Verify that the Diagnostic Viewer window in Simulink displays this message:

```
In the Simulation 3D Scene Configuration block, you set the scene
source to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the
scene.
```

This message confirms that Simulink has instantiated the vehicles and other assets in the Unreal Engine 3D environment.

c. In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor.

Observe the results in the To Video display window. The window displays the image from the camera.

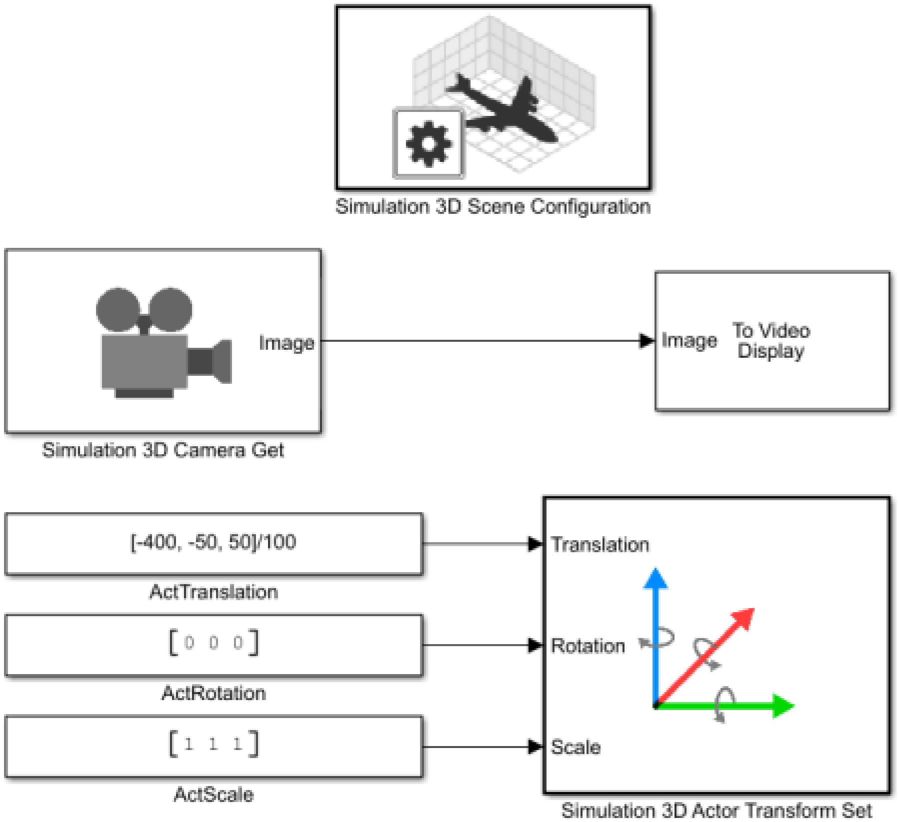


Place Camera on Vehicle in Custom Project

Follow these steps to create a custom Unreal Engine project and place a camera on a vehicle in the project. Although the example uses the To Video Display block from Computer Vision Toolbox, you can use a different visualization block to display the image.

To start, you need the Aerospace Blockset Interface for Unreal Engine Projects support package. See “Install Support Package and Configure Environment” on page 4-3.

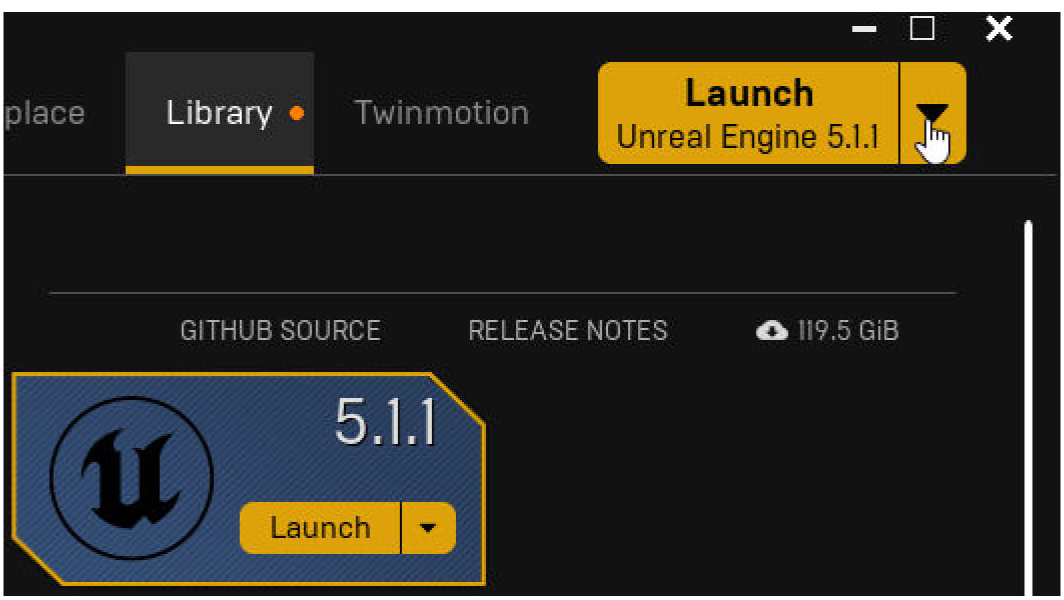
1. In a Simulink model, add the Simulation 3D Scene Configuration, Simulation 3D Camera Get, To Video Display, Simulation 3D Actor Transform Set, and three Constant blocks.



Save the model.

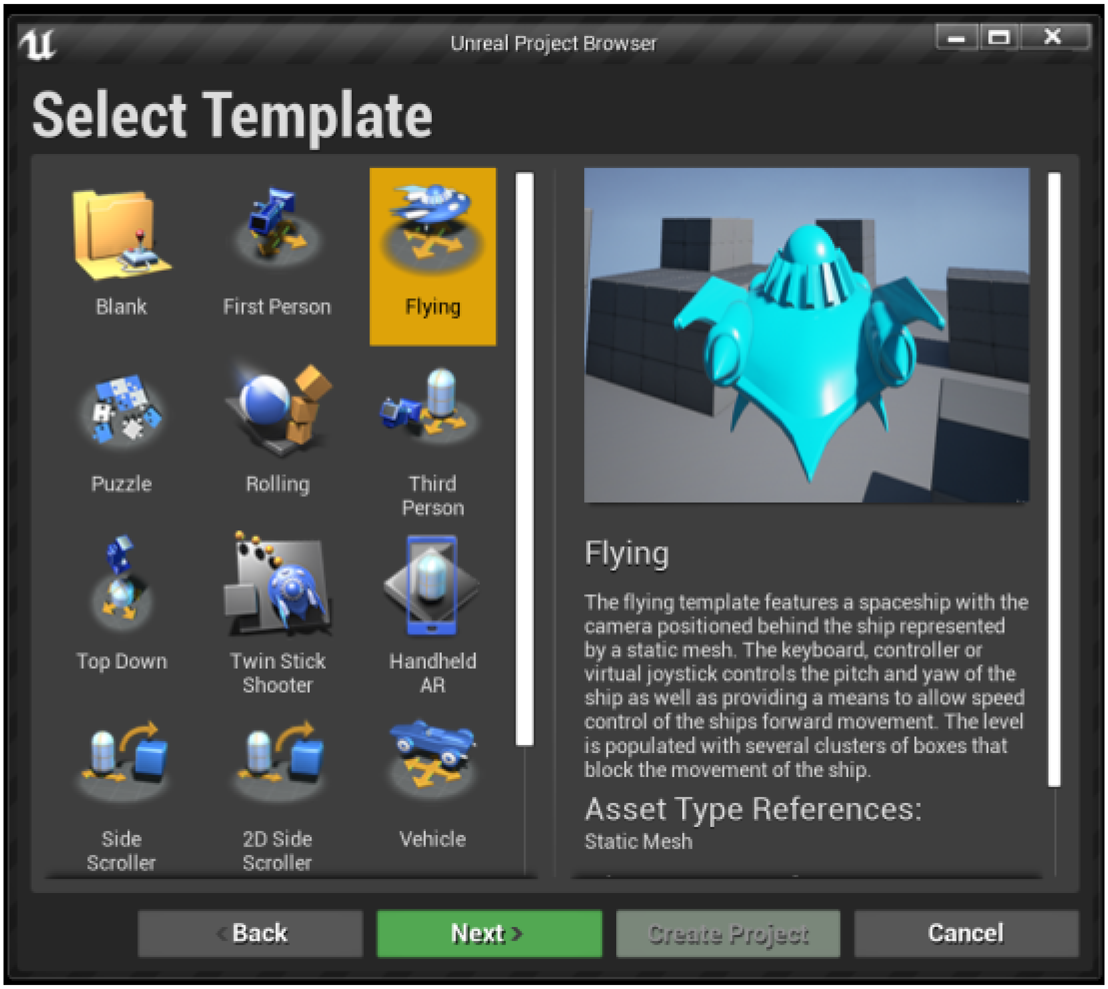
2 Create a new project using the **Flying** template from the Epic Games Launcher by Epic Games.

   a In the Epic Games Launcher, launch Unreal Engine 5.1.

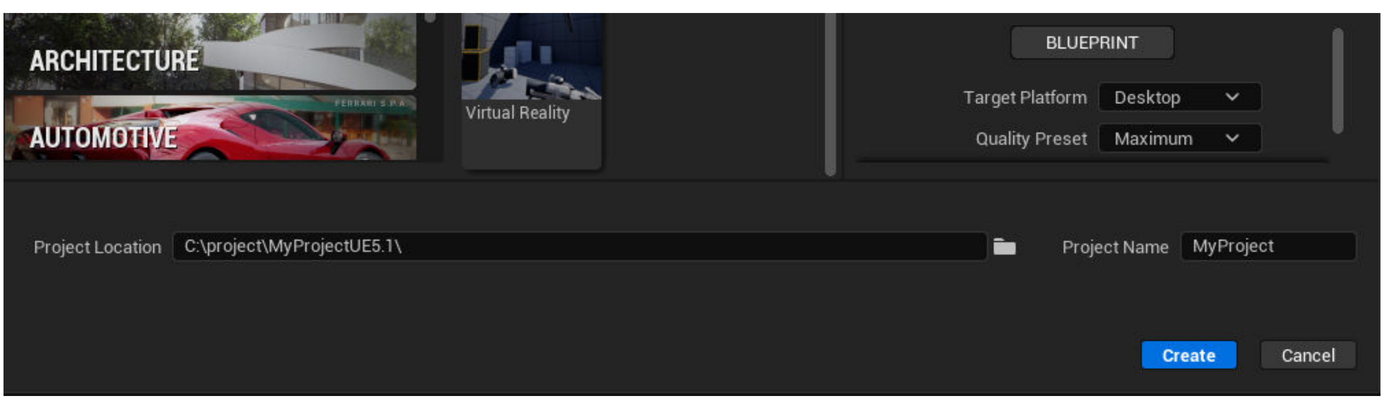


For more information about the Epic Games Launcher, see Unreal Engine.

b In the Unreal Project Browser, select **Games** and **Flying**.



c In the browser, create a Blueprint or C++ project. Provide a project name and location. Click **Create**.

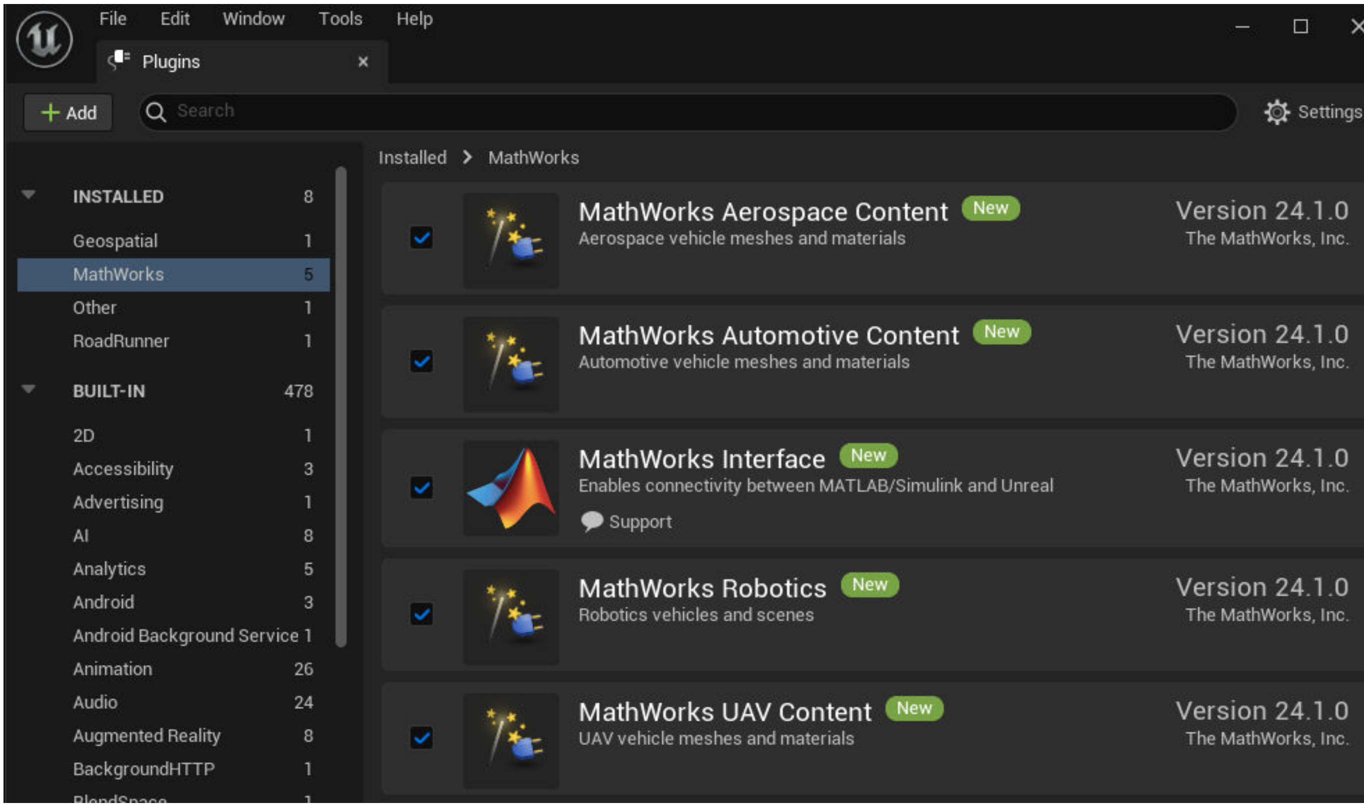


The Epic Games Launcher creates a new project and opens the Unreal Editor.

d Enable the MathWorks plugins.

   i Select **Edit** > **Plugins**.

   ii On the **Plugins** tab, navigate to MathWorks. Select **Enabled** for the content you want to enable, including MathWorks Interface.



e Save the project. Close the Unreal Editor.

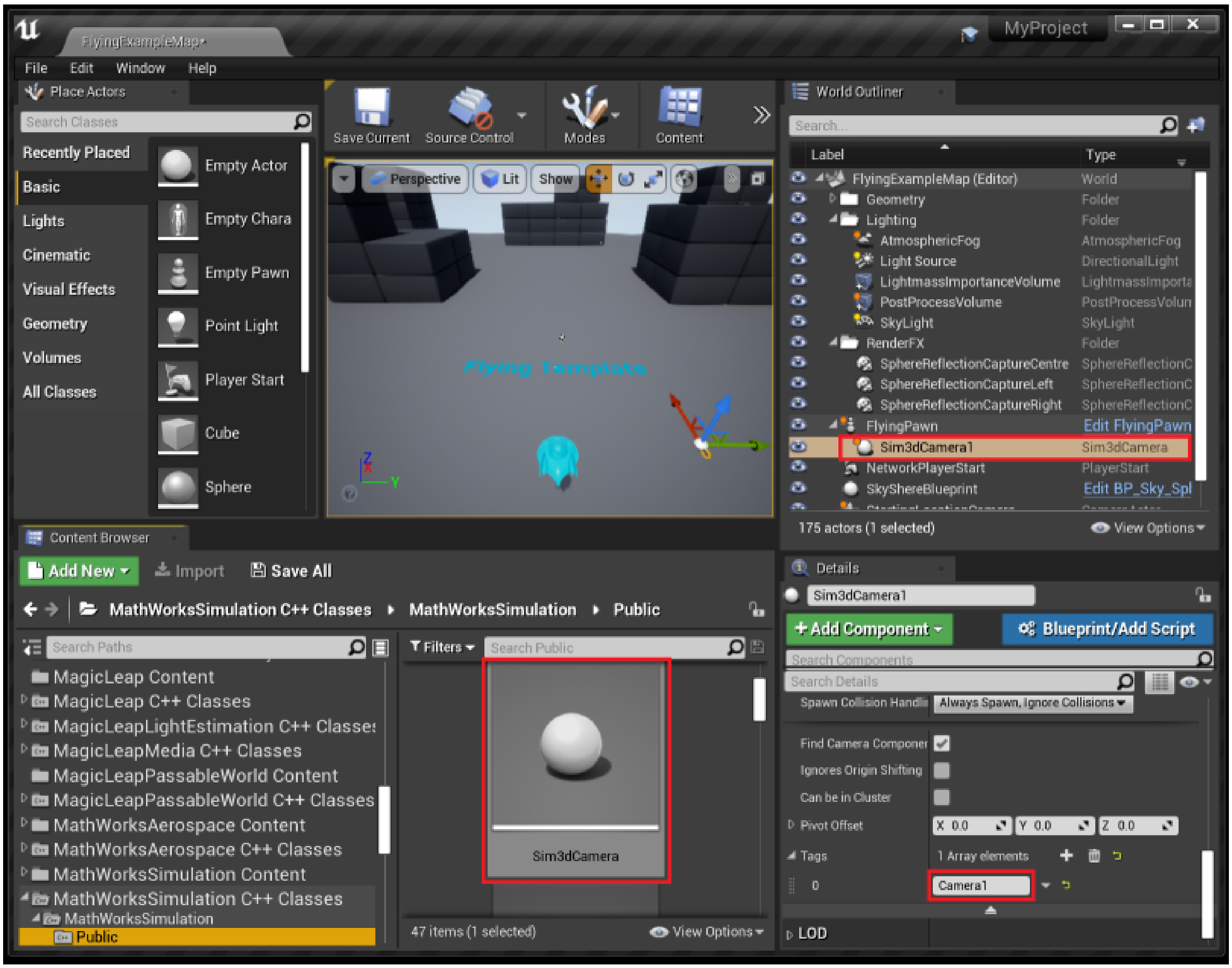
3 Open the Simulink model that you saved in step 1. Set these block parameters.

Block	Parameter Settings
Simulation 3D Scene Configuration	• **Scene Source** — `Unreal Editor` • **Project** — Specify the path an project that you saved in step 2. For example, `myProjectPath\myProject.uproject`
Simulation 3D Camera Get	• **Sensor identifier** — `1` • **Vehicle name** — `Scene Origin` • **Vehicle mounting location** — `Origin`
Simulation 3D Actor Transform Set	• **Tag for actor in 3D scene** — `Camera1`
ActTranslation	• **Constant value** — `[-400, -50, 50]/100` • **Interpret vector parameters as 1-D** — `off`

Block	Parameter Settings
ActRotation	• **Constant value** — `[0 0 0]` • **Interpret vector parameters as 1-D** — `off`
ActScale	• **Constant value** — `[1 1 1]` • **Interpret vector parameters as 1-D** — `off`

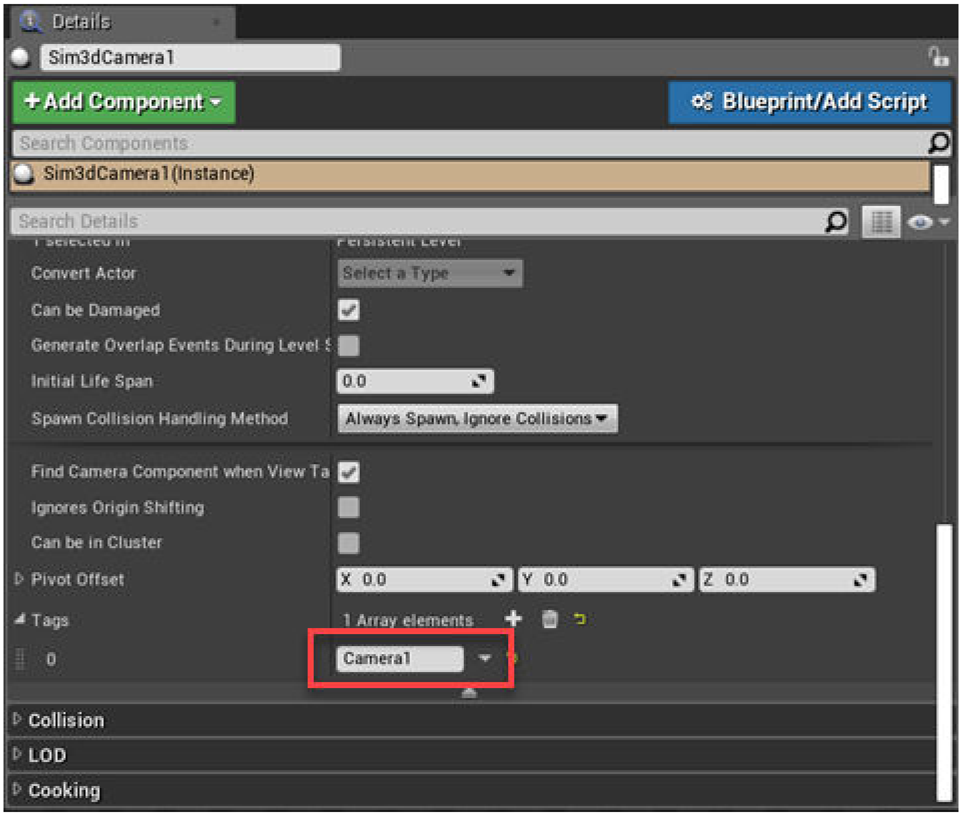
4 In the Simulation 3D Scene Configuration block, select **Open Unreal Editor**.

5 In the Unreal Editor, in the **Content Browser** navigate to `Sim3DCamera` under the `MathWorksSimulation C++` folder. Add it to the map by dragging it into the viewport.



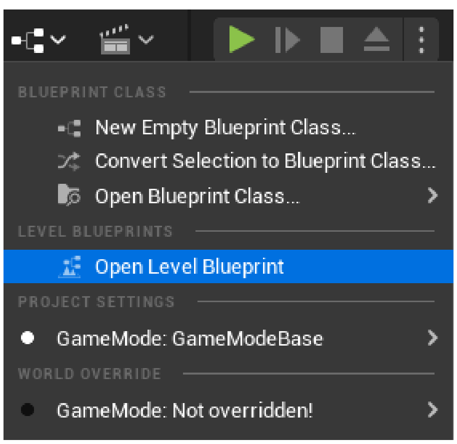
6 In World Outliner, drag and drop the camera onto the `FlyingPawn` blueprint.

7 On the **Details** tab, tag the `Sim3dCamera1` with the name `Camera1`.

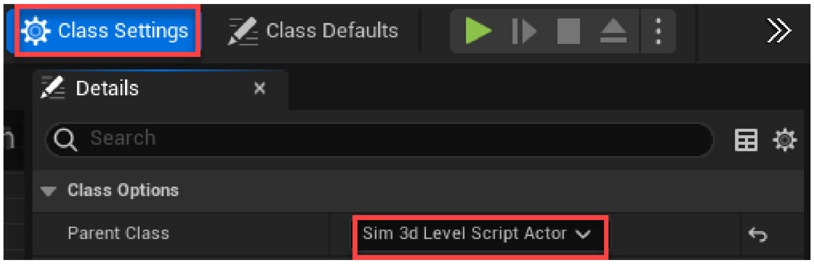


8 Set the parent class.

a Under **Blueprints**, click **Open Level Blueprint**, and select **Class Settings**.



b In the **Class Options**, set **Parent Class** to `Sim 3d Level Script Actor`.



9 Save the project.

10 Run the simulation.

a In the Simulink model, click **Run**.

Because the source of the scenes is the project opened in the Unreal Editor, the simulation does not start.

b Verify that the Diagnostic Viewer window in Simulink displays this message:

```
In the Simulation 3D Scene Configuration block, you set the scene
source to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the
scene.
```

This message confirms that Simulink has instantiated the vehicles and other assets in the Unreal Engine 3D environment.

c In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor.

See Also

Simulation 3D Camera Get | Simulation 3D Scene Configuration

More About

- "Create Empty Project in Unreal Engine" on page 4-51
- "Get Started Communicating with the Unreal Engine Visualization Environment" on page 4-18

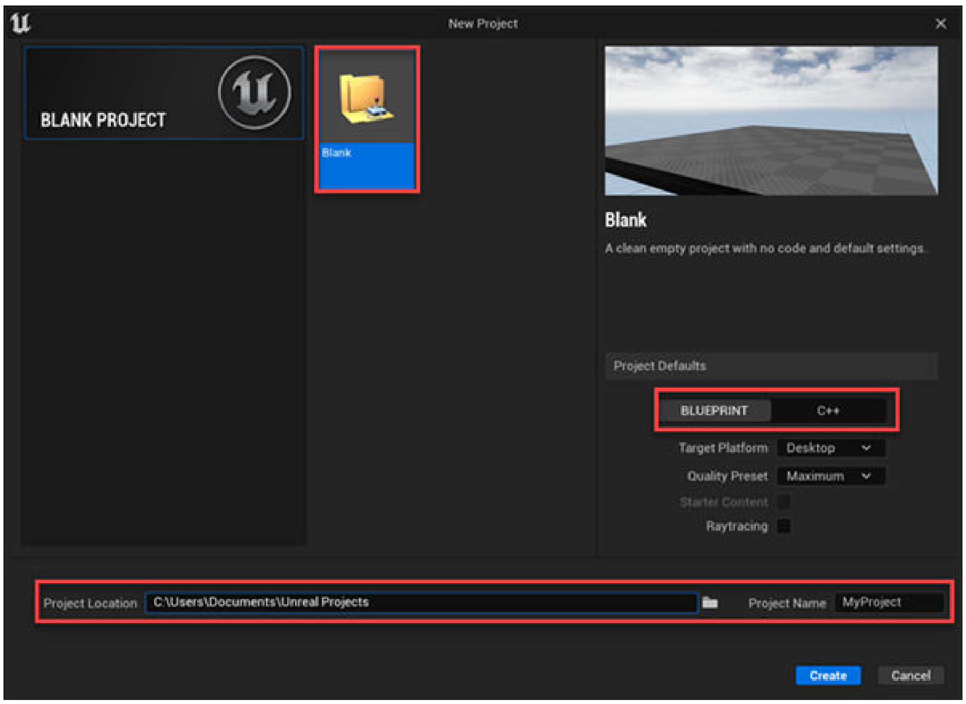
External Websites

- Unreal Engine

Create Empty Project in Unreal Engine

If you do not have an existing Unreal Engine project, you can create an empty project by following these steps.

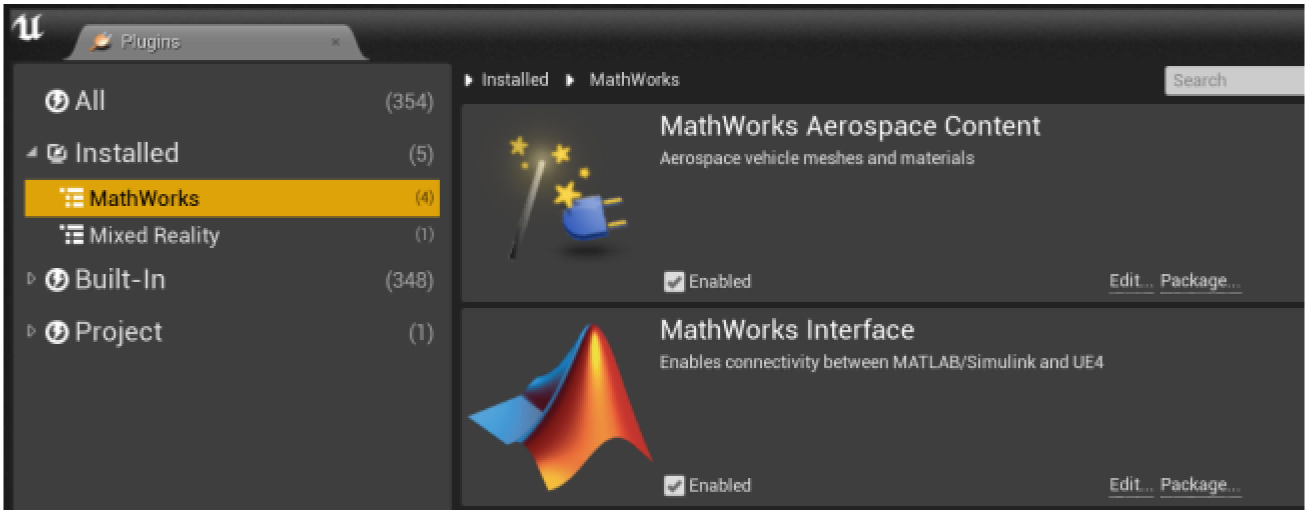
1. In Unreal Engine, select **File** > **New Project**.
2. Select a **Blank** template.
3. Under **Project Defaults**, select **BLUEPRINT** or **C++** to define the type of project to create.
4. Select a project name and location. Click **Create**.



5. Enable the MathWorks Interface plugin.

   a. Select **Edit** > **Plugins**.

   b. On the **Plugins** tab, navigate to MathWorks Interface. Select **Enabled**.



6 Save the project. Close the Unreal Editor.

7 Launch Simulink. In a Simulation 3D Scene Configuration block, set:

- **Scene Source** to `Unreal Editor`
- **Project** to the project created in step 6.

Select **Open Unreal Editor** to open the editor.

See Also

Simulation 3D Scene Configuration

More About

- "Build Light in Unreal Editor" on page 4-53
- "Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37

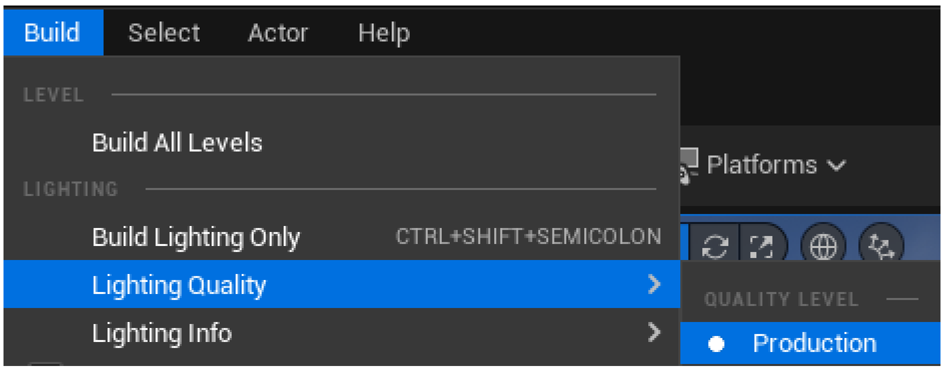
External Websites

- Unreal Engine

Build Light in Unreal Editor

Follow these steps to build light in the Unreal Editor. You can also use the `AutoVrtlEnv` project lighting in a custom scene.

1 In the editor, under **Build**, select **Lighting Quality** > **Production** to rebuild production quality maps. Rebuilding complex maps can be time-intensive.

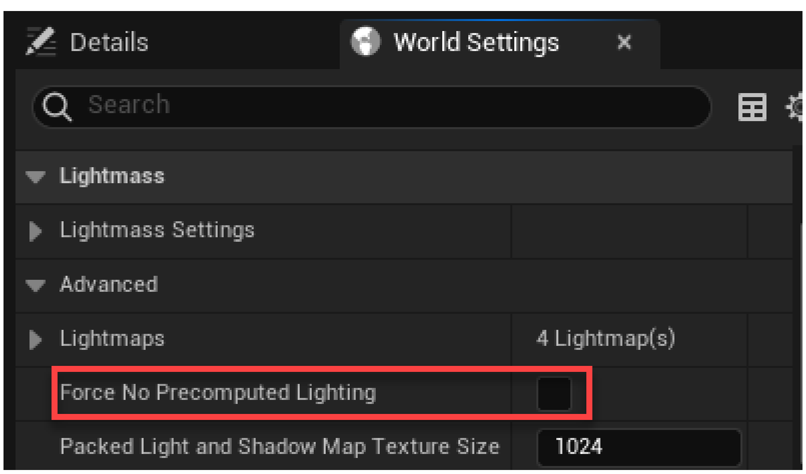


2 Click the **Build** icon to build the game. Production-quality lighting takes the a long time to build.

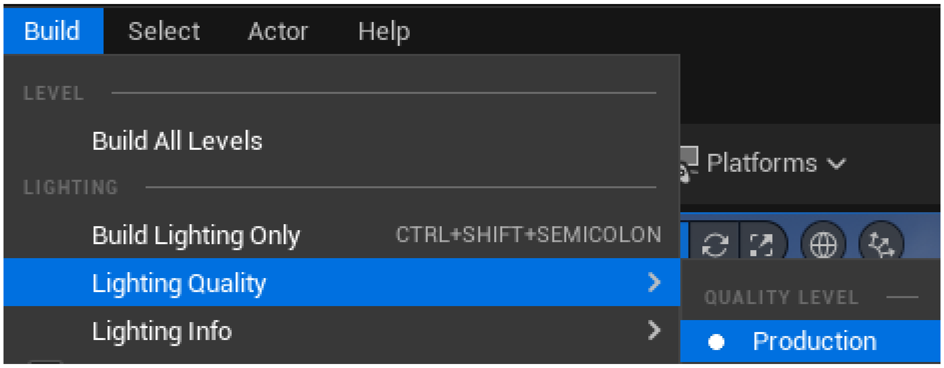
Use AutoVrtlEnv Project Lighting in Custom Scene

To use the lighting that comes installed with the `AutoVrtlEnv` project in Aerospace Blockset Interface for Unreal Engine Projects, follow these steps.

1 On the **World Settings** tab, clear **Force no precomputed lighting**.



2 Under **Build**, select **Lighting Quality** > **Production** to rebuild the maps with production quality. Rebuilding complex maps can be time-intensive.



See Also

Simulation 3D Scene Configuration

More About

- “Create Empty Project in Unreal Engine” on page 4-51
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37

External Websites

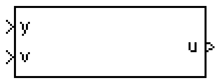
- Unreal Engine

5

Blocks

1D Controller [A(v),B(v),C(v),D(v)]

Implement gain-scheduled state-space controller depending on one scheduling parameter



Libraries:
Aerospace Blockset / GNC / Control

Description

The 1D Controller [A(v),B(v),C(v),D(v)] block implements a gain-scheduled state-space controller, as described in “Algorithms” on page 5-4.

The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

Output

u — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v) — *A* matrix of the state-space implementation

`A1` (default) | array

A-matrix of the state-space implementation, specified as an array. In the case of 1-D scheduling, the *A*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *A*-matrix corresponding to the first entry of *v* is the identity matrix, then `A(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A1'`

B-matrix(v) — *B* matrix of the state-space implementation

`B1` (default) | array

B-matrix of the state-space implementation, specified as an array. In the case of 1-D scheduling, the *B*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *B*-matrix corresponding to the first entry of *v* is the identity matrix, then `B(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B1'`

C-matrix(v) — *C* matrix of the state-space implementation

`C1` (default) | array

C-matrix of the state-space implementation, specified as a vector. In the case of 1-D scheduling, the *C*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *C*-matrix corresponding to the first entry of *v* is the identity matrix, then `C(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C1'`

D-matrix(v) — *D*

`D1` (default) | array

D-matrix of the state-space implementation, specified as an array. In the case of 1-D scheduling, the *D*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *D*-matrix corresponding to the first entry of *v* is the identity matrix, then `D(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector

Values: vector
Default: `'D1'`

Scheduling variable breakpoints — Breakpoints for scheduling variable

`v_vec` (default) | vector

Breakpoints for the scheduling variable, specified as a vector. The length of v must be the same as the size of the third dimension of A, B, C, and D.

Programmatic Use
Block Parameter: `AoA_vec`
Type: character vector
Values: vector
Default: `'v_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Initial states for the controller, such as initial values for the state vector, x, specified as a vector. The length of the vector must equal the size of the first dimension of A.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Algorithms

The block implements a gain-scheduled state-space controller as defined by this equation:

$$\dot{x} = A(v)x + B(v)y$$
$$u = C(v)x + D(v)y$$

where v is a parameter over which A, B, C, and D are defined. This type of controller scheduling assumes that the matrices A, B, C, and D vary smoothly as a function of v, which is often the case in aerospace applications.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

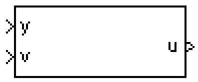
Generate C and C++ code using Simulink® Coder™.

See Also

1D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 1D Self-Conditioned [A(v),B(v),C(v),D(v)] | 2D Controller [A(v),B(v),C(v),D(v)] | 3D Controller [A(v),B(v),C(v),D(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

1D Controller Blend: u=(1-L).K1.y+L.K2.y

Implement 1-D vector of state-space controllers by linear interpolation of their outputs



Libraries:
Aerospace Blockset / GNC / Control

Description

The 1D Controller Blend u=(1-L).K1.y+L.K2.y block implements an array of state-space controller designs. The model runs the controllers in parallel and interpolates their outputs according to the current flight condition or operating point. The advantage of this implementation approach is that the state-space matrices *A*, *B*, *C*, and *D* for the individual controller designs do not need to vary smoothly from one design point to the next. The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

This block requires the Control System Toolbox™ license.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

Output

u — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v) — *A*-matrix of the state-space implementation

A1 (default) | array

A-matrix of the state-space implementation, specified as an array. In the case of 1-D blending, the *A*-matrix should have three dimensions, the last one corresponding to scheduling variable *v*. For example, if the *A*-matrix corresponding to the first entry of *v* is the identity matrix, then `A(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A1'`

B-matrix(v) — *B*-matrix of the state-space implementation

B1 (default) | array

B-matrix of the state-space implementation, specified as an array. In the case of 1-D scheduling, the *B*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *B*-matrix corresponding to the first entry of *v* is the identity matrix, then `B(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B1'`

C-matrix(v) — *C*-matrix of the state-space implementation

C1 (default) | array

C-matrix of the state-space implementation, specified as an array. In the case of 1-D scheduling, the *C*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *C*-matrix corresponding to the first entry of *v* is the identity matrix, then `C(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C1'`

D-matrix(v) — *D*-matrix of the state-space implementation

D1 (default) | array

D-matrix of the state-space implementation, specified as an array. In the case of 1-D scheduling, the *D*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *D*-matrix corresponding to the first entry of *v* is the identity matrix, then `D(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector
Values: vector
Default: `'D1'`

Scheduling variable breakpoints — Breakpoints for scheduling variable

`[1 1.5 2]` (default) | vector

Breakpoints for the scheduling variable, specified as a vector. The length of *v* must be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v`
Type: character vector
Values: vector
Default: `'[1 1.5 2]'`

Initial state, x_initial — Initial states

`0` (default) | vector

Initial states for the controller, such as initial values for the state vector, *x*, specified as a vector. The length must equal the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Poles of A(v)-H(v)*C(v) = [w1 ... wn]) — Poles of observer

`[-5 -2]` (default) | vector

Poles of observer, specified as a vector. For incoming controllers, the block uses an observer-like structure to ensure that the controller output tracks the current block output, *u*. The number of poles must equal the dimension of the *A*-matrix. Poles that are too fast result in sensor noise propagation; poles that are too slow result in the failure of the controller output to track *u*.

Programmatic Use
Block Parameter: `vec_w`
Type: character vector
Values: vector
Default: `'[-5 -2]'`

Algorithms

The block implements

$$\dot{x}_1 = A_1 x_1 + B_1 y$$
$$u_1 = C_1 x_1 + D_1 y$$
$$\dot{x}_2 = A_2 x_2 + B_2 y$$
$$u_2 = C_2 x_2 + D_2 y$$
$$u = (1 - \lambda) u_1 + \lambda u_2$$
$$\lambda = \begin{cases} 0 & v < v_{\min} \\ \dfrac{v - v_{\min}}{v_{\max} - v_{\min}} & v_{\min} \le v \le v_{\max} \\ 1 & v > v_{\max} \end{cases}$$

For example, suppose two controllers are designed at two operating points $v = v_{\min}$ and $v = v_{\max}$. For longer arrays of design points, the block only implements nearest neighbor designs. At any given instant in time, the block updates three controller designs, reducing computational requirements.

As the value of the scheduling parameter varies and the index of the controllers that need to be run changes, the block initializes the states of the oncoming controller using the self-conditioned form as defined for the Self-Conditioned [A,B,C,D] block.

Version History

Introduced before R2006a

References

[1] Hyde, R. A., "H-infinity Aerospace Control Design — A VSTOL Flight Application." , *Advances in Industrial Control Series*, Springer Verlag, 1995.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 1D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 1D Self-Conditioned [A(v),B(v),C(v),D(v)] | 2D Controller Blend | Self-Conditioned [A,B,C,D] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

1D Observer Form [A(v),B(v),C(v),F(v),H(v)]

Implement gain-scheduled state-space controller in observer form depending on one scheduling parameter

Libraries:
Aerospace Blockset / GNC / Control

Description

The 1D Observer Form [A(v),B(v),C(v),F(v),H(v)] block implements a gain-scheduled state-space controller as defined in “Algorithms” on page 5-12.

The output from this block is the actuator demand, which you can input to an actuator block. Use this block to implement a controller designed using *H*-infinity loop-shaping, one of the design methods supported by Robust Control Toolbox.

Limitations

If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.

Ports

Input

y-y_dem — Set-point error
vector

Set-point error, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

u_meas — Measured actuator position
vector

Measured actuator position, specified as a vector.

Data Types: `double`

Output

u_dem — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. The *A*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *A*-matrix corresponding to the first entry of *v* is the identity matrix, then `A(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v) — *B*-matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. The *B*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *B*-matrix corresponding to the first entry of *v* is the identity matrix, then `B(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v) — *C*-matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. The *C*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. Hence, for example, if the *C*-matrix corresponding to the first entry of *v* is the identity matrix, then `C(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

F-matrix(v) — *F*-matrix of the state-space implementation

`F` (default) | array

State-feedback matrix. The F-matrix should have three dimensions, the last one corresponding to the scheduling variable v. Hence, for example, if the F-matrix corresponding to the first entry of v is the identity matrix, then `F(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: vector
Default: `'F'`

H-matrix(v) — H-matrix of the state-space implementation

`H` (default) | array

Observer (output injection) matrix. The H-matrix should have three dimensions, the last one corresponding to the scheduling variable v. Hence, for example, if the H-matrix corresponding to the first entry of v is the identity matrix, then `H(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `H`
Type: character vector
Values: vector
Default: `'H'`

Scheduling variable breakpoints — Breakpoints for scheduling variable

`v_vec` (default) | vector

Breakpoints for the scheduling variable, specified as a vector. The length of v should be same as the size of the third dimension of A, B, C, F, and H.

Programmatic Use
Block Parameter: `AoA_vec`
Type: character vector
Values: vector
Default: `'v_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Initial states for the controller, i.e., initial values for the state vector, x, specified as a vector. It should have length equal to the size of the first dimension of A.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Algorithms

The block implements a gain-scheduled state-space controller defined in the following observer form:

$$\dot{x} = (A(v) + H(v)C(v))x + B(v)u_{meas} + H(v)(y - y_{dem})$$
$$u_{dem} = F(v)x$$

Version History

Introduced before R2006a

References

[1] Hyde, R. A., "H-infinity Aerospace Control Design — A VSTOL Flight Application," Springer Verlag, *Advances in Industrial Control Series*, 1995.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 1D Controller Blend: u=(1-L).K1.y+L.K2.y | 1D Self-Conditioned [A(v),B(v),C(v),D(v)] | 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 3D Observer Form [A(v),B(v),C(v),F(v),H(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

1D Self-Conditioned [A(v),B(v),C(v),D(v)]

Implement gain-scheduled state-space controller in self-conditioned form depending on one scheduling parameter

Libraries:
Aerospace Blockset / GNC / Control

Description

The 1D Self-Conditioned [A(v),B(v),C(v),D(v)] block implements a gain-scheduled state-space controller as defined in “Algorithms” on page 5-16.

The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

- If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.
- This block requires the Control System Toolbox license.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v — Scheduling variable
vector

Scheduling variable, specified as a vector, ordered according to the dimensions of the state-space matrices.

Data Types: `double`

u_meas — Measured actuator position
vector

Measured actuator position, specified as a vector.

Data Types: `double`

Output

u_dem — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v) — *A*-matrix of the state-space implementation

`A` (default) | array

A-matrix of the state-space implementation. The *A*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *A*-matrix corresponding to the first entry of *v* is the identity matrix, then `A(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v) — *B*-matrix of the state-space implementation

`B` (default) | array

B-matrix of the state-space implementation. The *B*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *B*-matrix corresponding to the first entry of *v* is the identity matrix, then `B(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v) — *C*-matrix of the state-space implementation

`C` (default) | array

C-matrix of the state-space implementation. The *C*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *C*-matrix corresponding to the first entry of *v* is the identity matrix, then `C(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

D-matrix(v) — *D*-matrix of the state-space implementation

`D` (default) | array

D-matrix of the state-space implementation. The *D*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *D*-matrix corresponding to the first entry of *v* is the identity matrix, then `D(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector
Values: vector
Default: `'D'`

Scheduling variable breakpoints — Breakpoints for scheduling variable

`v_vec` (default) | vector

Vector of the breakpoints for the first scheduling variable. The length of *v* should be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v`
Type: character vector
Values: vector
Default: `'v_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Vector of initial states for the controller, that is, initial values for the state vector, *x*. It should have length equal to the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Poles of A(v)-H(v)*C(v) — Desired poles

`[-5 -2]` (default) | vector

Desired poles of *A*-*HC*, specified as a vector. The poles are assigned to the same locations for all values of the scheduling parameter *v*. Hence, the number of pole locations defined should be equal to the length of the first dimension of the *A*-matrix.

Programmatic Use
Block Parameter: `vec_w`
Type: character vector
Values: vector
Default: `'[-5 -2]'`

Algorithms

The block implements a gain-scheduled state-space controller as defined by the equations:

$$\dot{x} = A(v)x + B(v)y$$
$$u = C(v)x + D(v)y$$

in the self-conditioned form

$$\dot{z} = (A(v) - H(v)C(v))z + (B(v) - H(v)D(V))e + H(v)u_{meas}$$
$$u_{dem} = C(v)z + D(v)e$$

This block implements a gain-scheduled version of the Self-Conditioned [A,B,C,D] block, where *v* is the parameter over which *A*, *B*, *C*, and *D* are defined. This type of controller scheduling assumes that the matrices *A*, *B*, *C*, and *D* vary smoothly as a function of *v*, which is often the case in aerospace applications.

Version History

Introduced before R2006a

References

[1] Kautsky, Nichols, and Van Dooren. "Robust Pole Assignment in Linear State Feedback." *International Journal of Control*, Vol. 41, Number 5, 1985, pp. 1129-1155.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 1D Controller Blend: u=(1-L).K1.y+L.K2.y | 1D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | 3D Self-Conditioned [A(v),B(v),C(v),D(v)] | Self-Conditioned [A,B,C,D] | Self-Conditioned [A,B,C,D] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

2D Controller [A(v),B(v),C(v),D(v)]

Implement gain-scheduled state-space controller depending on two scheduling parameters

Libraries:
Aerospace Blockset / GNC / Control

Description

The 2D Controller [A(v),B(v),C(v),D(v)] block implements a gain-scheduled state-space controller, as described in “Algorithms” on page 5-20.

The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v1 — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

Output

u — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 2-D scheduling, the *A*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *A*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `A(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2) — *B*-matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. In the case of 2-D scheduling, the *B*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *B*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `B(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2) — -matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. In the case of 2-D scheduling, the *C*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *C*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `C(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

D-matrix(v1,v2) — *D*-matrix of the state-space implementation

D (default) | array

D-matrix of the state-space implementation. In the case of 2-D scheduling, the *D*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *D*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `D(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector
Values: vector
Default: `'D'`

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

`v1_vec` (default) | vector

Vector of the breakpoints for the first scheduling variable. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `AoA_vec`
Type: character vector
Values: vector
Default: `'v1_vec'`

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

`v2_vec` (default) | vector

Vector of the breakpoints for the second scheduling variable. The length of *v*2 should be same as the size of the fourth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `Mach_vec`
Type: character vector
Values: vector
Default: `'v2_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Vector of initial states for the controller, that is, initial values for the state vector, *x*. It should have length equal to the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Algorithms

The block implements a gain-scheduled state-space controller as defined by this equation:

$$\dot{x} = A(v)x + B(v)y$$
$$u = C(v)x + D(v)y$$

where *v* is a vector of parameters over which *A*, *B*, *C*, and *D* are defined. This type of controller scheduling assumes that the matrices *A*, *B*, *C*, and *D* vary smoothly as a function of *v*, which is often the case in aerospace applications.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

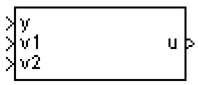
Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 2D Controller Blend | 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | 3D Controller [A(v),B(v),C(v),D(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

2D Controller Blend

Implement 2-D vector of state-space controllers by linear interpolation of their outputs



Libraries:
Aerospace Blockset / GNC / Control

Description

The 2D Controller Blend block implements an array of state-space controller designs. The controllers are run in parallel, and their outputs interpolated according to the current flight condition or operating point. The advantage of this implementation approach is that the state-space matrices *A*, *B*, *C*, and *D* for the individual controller designs do not need to vary smoothly from one design point to the next. The output from this block is the actuator demand, which you can input to an actuator block.

For the 2D Controller Blend block, at any given instant in time, nine controller designs are updated.

As the value of the scheduling parameter varies and the index of the controllers that need to be run changes, the states of the oncoming controller are initialized by using the self-conditioned form as defined for the Self-Conditioned [A,B,C,D] block.

Limitations

This block requires the Control System Toolbox license.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v1 — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Scheduling variable
vector

Scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

Output

u — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 2-D blending, the *A*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *A*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `A(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2) — *B*-matrix of the state-space implementation

A (default) | array

B-matrix of the state-space implementation. The *B*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *B*-matrix corresponding to the first entry of *v* is the identity matrix, then `B(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2) — *C*-matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. The *C*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *C*-matrix corresponding to the first entry of *v* is the identity matrix, then `C(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

D-matrix(v1,v2) — *D*-matrix of the state-space implementation

C (default) | array

D-matrix of the state-space implementation. The *D*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *D*-matrix corresponding to the first entry of *v* is the identity matrix, then `D(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector
Values: vector
Default: `'D'`

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

`v1_vec` (default) | vector

Breakpoints for the first scheduling variable, specified as a vector. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v1`
Type: character vector
Values: vector
Default: `'v1_vec'`

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

`v2_vec` (default) | vector

Breakpoints for the second scheduling variable, specified as a vector. The length of *v*2 should be same as the size of the fourth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v2`
Type: character vector
Values: vector
Default: `'v2_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Vector of initial states for the controller, that is, initial values for the state vector, *x*. It should have length equal to the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Poles of A(v)-H(v)*C(v) — Desired poles

`[-5 -2]` (default)

For oncoming controllers, an observer-like structure is used to ensure that the controller output tracks the current block output, *u*. The poles of the observer are defined in this dialog box as a vector, the number of poles being equal to the dimension of the *A*-matrix. Poles that are too fast result in sensor noise propagation, and poles that are too slow result in the failure of the controller output to track *u*.

Programmatic Use
Block Parameter: `vec_w`
Type: character vector
Values: vector
Default: `'[-5 -2]'`

Version History

Introduced before R2006a

References

[1] Hyde, R. A. "H-infinity Aerospace Control Design - A VSTOL Flight Application." Springer Verlag: *Advances in Industrial Control Series*, 1995.

Extended Capabilities

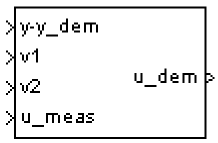
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller Blend: u=(1-L).K1.y+L.K2.y | 2D Controller [A(v),B(v),C(v),D(v)] | 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | Self-Conditioned [A,B,C,D] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

2D Observer Form [A(v),B(v),C(v),F(v),H(v)]

Implement gain-scheduled state-space controller in observer form depending on two scheduling parameters



Libraries:
Aerospace Blockset / GNC / Control

Description

The 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] block implements a gain-scheduled state-space controller as defined in “Algorithms” on page 5-29.

The output from this block is the actuator demand, which you can input to an actuator block. Use this block to implement a controller designed using *H*-infinity loop-shaping, one of the design methods supported by Robust Control Toolbox.

Limitations

If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.

Ports

Input

y-y_dem — Set-point error
vector

Set-point error, specified as a vector.

Data Types: `double`

v1 — First scheduling variable
vector

First scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Second scheduling variable
vector

Second scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

u_meas — Measured actuator position
vector

Measured actuator position, specified as a vector.

Data Types: `double`

Output

u_dem — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 2-D scheduling, the *A*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *A*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `A(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2) — *B*-matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. In the case of 2-D scheduling, the *B*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *B*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `B(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2) — *C*-matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. In the case of 2-D scheduling, the *C*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *C*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `C(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

F-matrix(v1,v2) — *F*-matrix of the state-space implementation

`F` (default) | array

State-feedback matrix. In the case of 2-D scheduling, the *F*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *F*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `F(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: vector
Default: `'F'`

H-matrix(v1,v2) — *H*-matrix of the state-space implementation

`H` (default) | array

Observer (output injection) matrix. In the case of 2-D scheduling, the *H*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *H*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `H(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `H`
Type: character vector
Values: vector
Default: `'H'`

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

`v1_vec` (default)

Vector of the breakpoints for the first scheduling variable. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, *F*, and *H*.

Programmatic Use
Block Parameter: `AoA_vec`
Type: character vector
Values: vector
Default: `'v1_vec'`

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

`v2_vec` (default)

Vector of the breakpoints for the second scheduling variable. The length of *v*2 should be same as the size of the fourth dimension of *A*, *B*, *C*, *F*, and *H*.

Programmatic Use

Block Parameter: `Mach_vec`
Type: character vector
Values: vector
Default: `'v2_vec'`

Initial state, x_initial — Initial states

`0` (default)

Vector of initial states for the controller,that is, initial values for the state vector, x. It should have length equal to the size of the first dimension of A.

Programmatic Use

Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Algorithms

The block implements a gain-scheduled state-space controller defined in the following observer form:

$$\dot{x} = (A(v) + H(v)C(v))x + B(v)u_{meas} + H(v)(y - y_{dem})$$

$$u_{dem} = F(v)x$$

Version History

Introduced before R2006a

References

[1] Hyde, R. A.. "H-infinity Aerospace Control Design — A VSTOL Flight Application." *Advances in Industrial Control Series*, Springer Verlag, 1995.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 2D Controller [A(v),B(v),C(v),D(v)] | 2D Controller Blend | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | 3D Observer Form [A(v),B(v),C(v),F(v),H(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

2D Self-Conditioned [A(v),B(v),C(v),D(v)]

Implement gain-scheduled state-space controller in self-conditioned form depending on two scheduling parameters

Libraries:
Aerospace Blockset / GNC / Control

Description

The 2D Self-Conditioned [A(v),B(v),C(v),D(v)] block implements a gain-scheduled state-space controller as defined in “Algorithms” on page 5-33.

The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

- If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.
- This block requires the Control System Toolbox license.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v1 — First scheduling variable
vector

First scheduling variable, specified as a vector, ordered according to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Second scheduling variable
vector

Second scheduling variable, specified as a vector, ordered according to the dimensions of the state-space matrices.

Data Types: `double`

u_meas — Measured actuator position
vector

Measured actuator position, specified as a vector.

Data Types: `double`

Output

u_dem — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 2-D scheduling, the *A*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *A*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `A(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2) — *B*-matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. In the case of 2-D scheduling, the *B*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *B*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `B(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2) — *C*-matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. In the case of 2-D scheduling, the *C*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *C*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `C(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

D-matrix(v1,v2) — *D*-matrix of the state-space implementation

`D` (default) | array

D-matrix of the state-space implementation. In the case of 2-D scheduling, the *D*-matrix should have four dimensions, the last two corresponding to scheduling variables *v*1 and *v*2. For example, if the *D*-matrix corresponding to the first entry of *v*1 and first entry of *v*2 is the identity matrix, then `D(:,:,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector
Values: vector
Default: `'D'`

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

v1_vec (default) | vector

Vector of the breakpoints for the first scheduling variable. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v1`
Type: character vector
Values: vector
Default: `'v1_vec'`

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

v2_vec (default) | vector

Vector of the breakpoints for the second scheduling variable. The length of *v*2 should be same as the size of the fourth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v2`
Type: character vector
Values: vector
Default: `'v2_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Vector of initial states for the controller, that is, initial values for the state vector, *x*. It should have length equal to the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Poles of A(v)-H(v)*C(v) — Desired poles

`[-5 -2]` (default) | vector

Vector of the desired poles of *A-HC*. Note that the poles are assigned to the same locations for all values of the scheduling parameter, *v*. Hence, the number of pole locations defined should be equal to the length of the first dimension of the *A*-matrix.

Programmatic Use
Block Parameter: `vec_w`
Type: character vector
Values: vector
Default: `'[-5 -2]'`

Algorithms

The block implements a gain-scheduled state-space controller as defined by the equations:

$$\dot{x} = A(v)x + B(v)y$$
$$u = C(v)x + D(v)y$$

in the self-conditioned form

$$\dot{z} = (A(v) - H(v)C(v))z + (B(v) - H(v)D(v))e + H(v)u_{meas}$$
$$u_{dem} = C(v)z + D(v)e$$

For the rationale behind this self-conditioned implementation, refer to the Self-Conditioned [A,B,C,D] block reference. This block implements a gain-scheduled version of the Self-Conditioned [A,B,C,D] block, *v* being the vector of parameters over which *A*, *B*, *C*, and *D* are defined. This type of controller scheduling assumes that the matrices *A*, *B*, *C*, and *D* vary smoothly as a function of *v*, which is often the case in aerospace applications.

Version History
Introduced before R2006a

References

[1] Kautsky, Nichols, and Van Dooren. "Robust Pole Assignment in Linear State Feedback," *International Journal of Control*, Vol. 41, Number 5, 1985, pp 1129-1155.

Extended Capabilities

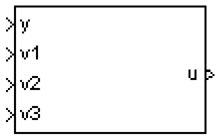
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

1D Self-Conditioned [A(v),B(v),C(v),D(v)] | 2D Controller [A(v),B(v),C(v),D(v)] | 2D Controller Blend | 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 3D Self-Conditioned [A(v),B(v),C(v),D(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

3D Controller [A(v),B(v),C(v),D(v)]

Implement gain-scheduled state-space controller depending on three scheduling parameters



Libraries:
Aerospace Blockset / GNC / Control

Description

The 3D Controller [A(v),B(v),C(v),D(v)] block implements a gain-scheduled state-space controller as described in “Algorithms” on page 5-38.

The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v1 — First scheduling variable
vector

First scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Second scheduling variable
vector

Second scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v3 — Third scheduling variable
vector

Second scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

Output

u — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2,v3) — *A* matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 3-D scheduling, the *A*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *A*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `A(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2,v3) — *B* matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. In the case of 3-D scheduling, the *B*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *B*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `B(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2,v3) — *C* matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. In the case of 3-D scheduling, the *C*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *C*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `C(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: C
Type: character vector
Values: vector
Default: 'C'

D-matrix(v1,v2,v3) — *D* matrix of the state-space implementation

D (default) | array

D-matrix of the state-space implementation. In the case of 3-D scheduling, the *D*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *D*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `D(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: D
Type: character vector
Values: vector
Default: 'D'

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

v1_vec (default) | vector

Vector of the breakpoints for the first scheduling variable. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: AoA_vec
Type: character vector
Values: vector
Default: 'v1_vec'

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

v2_vec (default) | vector

Vector of the breakpoints for the second scheduling variable. The length of *v*2 should be same as the size of the fourth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: AoS_vec
Type: character vector
Values: vector
Default: 'v2_vec'

Third scheduling variable (v3) breakpoints — Breakpoints for third scheduling variable

v3_vec (default) | vector

Vector of the breakpoints for the third scheduling variable. The length of *v*3 should be same as the size of the fifth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `Mach_vec`
Type: character vector
Values: vector
Default: `'v3_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Vector of initial states for the controller, i.e., initial values for the state vector, x. It should have length equal to the size of the first dimension of A.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Algorithms

The block implements a gain-scheduled state-space controller as defined by this equation:

$$\dot{x} = A(v)x + B(v)y$$
$$u = C(v)x + D(v)y$$

where v is a vector of parameters over which A, B, C, and D are defined. This type of controller scheduling assumes that the matrices A, B, C, and D vary smoothly as a function of v, which is often the case in aerospace applications.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

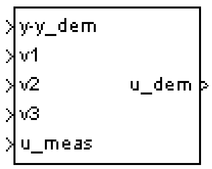
Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 2D Controller [A(v),B(v),C(v),D(v)] | 3D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 3D Self-Conditioned [A(v),B(v),C(v),D(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

3D Observer Form [A(v),B(v),C(v),F(v),H(v)]

Implement gain-scheduled state-space controller in observer form depending on three scheduling parameters



Libraries:
Aerospace Blockset / GNC / Control

Description

The 3D Observer Form [A(v),B(v),C(v),F(v),H(v)] block implements a gain-scheduled state-space controller defined in "Algorithms" on page 5-29.

The main application of this block is to implement a controller designed using H-infinity loop-shaping. Use this block to implement a controller designed using *H*-infinity loop-shaping, one of the design methods supported by Robust Control Toolbox.

Limitations

If the scheduling parameter inputs to the block go out of range, they are clipped. The state-space matrices are not interpolated out of range.

Ports

Input

y-y_dem — Set-point error
vector

Set-point error, specified as a vector.

Data Types: `double`

v1 — First scheduling variable
vector

First scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Second scheduling variable
vector

Second scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

v3 — Third scheduling variable
vector

Third scheduling variable, specified as a vector, that conforms to the dimensions of the state-space matrices.

Data Types: `double`

u_meas — Measured actuator position
vector

Measured actuator position, specified as a vector.

Data Types: `double`

Output

u_dem — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2,v3) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 3-D scheduling, the *A*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *A*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `A(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2,v3) — *B*-matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. In the case of 3-D scheduling, the *B*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *B*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `B(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2,v3) — *C*-matrix of the state-space implementation

`C` (default) | array

C-matrix of the state-space implementation. In the case of 3-D scheduling, the *C*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *C*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `C(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

F-matrix(v1,v2,v3) — *F*-matrix of the state-space implementation

`F` (default) | array

State-feedback matrix. In the case of 3-D scheduling, the *F*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *F*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `F(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: vector
Default: `'F'`

H-matrix(v1,v2,v3) — *H*-matrix of the state-space implementation

`H` (default) | array

Observer (output injection) matrix. In the case of 3-D scheduling, the *H*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *H*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `H(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `H`
Type: character vector
Values: vector
Default: `'H'`

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

`v1_vec` (default)

Vector of the breakpoints for the first scheduling variable. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, *F*, and *H*.

Programmatic Use
Block Parameter: `AoA_vec`
Type: character vector

Values: vector
Default: `'v1_vec'`

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

`v2_vec` (default)

Vector of the breakpoints for the second scheduling variable. The length of $v2$ should be same as the size of the fourth dimension of A, B, C, F, and H.

Programmatic Use
Block Parameter: `AoS_vec`
Type: character vector
Values: vector
Default: `'v2_vec'`

Third scheduling variable (v3) breakpoints — Breakpoints for third scheduling variable

`v3_vec` (default)

Vector of the breakpoints for the third scheduling variable. The length of $v3$ should be same as the size of the fifth dimension of A, B, C, F, and H.

Programmatic Use
Block Parameter: `Mach_vec`
Type: character vector
Values: vector
Default: `'v3_vec'`

Initial state, x_initial — Initial states

`0` (default)

Vector of initial states for the controller, that is, initial values for the state vector, x. It should have length equal to the size of the first dimension of A.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Algorithms

The block implements gain-scheduled state-space controller as defined by these equations:

$$\dot{x} = (A(v) + H(v)C(v))x + B(v)u_{meas} + H(v)(y - y_{dem})$$
$$u_{dem} = F(v)x$$

Version History

Introduced before R2006a

References

[1] Hyde, R. A. "H-infinity Aerospace Control Design — A VSTOL Flight Application." *Advances in Industrial Control Series*, Springer Verlag, 1995.

Extended Capabilities

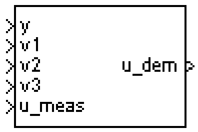
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

1D Controller [A(v),B(v),C(v),D(v)] | 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 3D Controller [A(v),B(v),C(v),D(v)] | 3D Self-Conditioned [A(v),B(v),C(v),D(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

3D Self-Conditioned [A(v),B(v),C(v),D(v)]

Implement gain-scheduled state-space controller in self-conditioned form depending on two scheduling parameters



Libraries:
Aerospace Blockset / GNC / Control

Description

The 3D Self-Conditioned [A(v),B(v),C(v),D(v)] block implements a gain-scheduled state-space controller as defined in “Algorithms” on page 5-47.

If the scheduling parameter inputs to the block go out of range, then they are clipped. The state-space matrices are not interpolated out of range.

The output from this block is the actuator demand, which you can input to an actuator block.

Limitations

This block requires the Control System Toolbox license.

Ports

Input

y — Aircraft measurements
vector

Aircraft measurements, specified as a vector.

Data Types: `double`

v1 — First scheduling variable
vector

First scheduling variable, specified as a vector, ordered according to the dimensions of the state-space matrices.

Data Types: `double`

v2 — Second scheduling variable
vector

Second scheduling variable, specified as a vector, ordered according to the dimensions of the state-space matrices.

Data Types: `double`

v3 — Third scheduling variable
vector

Third scheduling variable, specified as a vector, ordered according to the dimensions of the state-space matrices.

Data Types: `double`

u_meas — Measured actuator position
vector

Measured actuator position, specified as a vector.

Data Types: `double`

Output

Port_1 — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix(v1,v2,v3) — *A*-matrix of the state-space implementation

A (default) | array

A-matrix of the state-space implementation. In the case of 3-D scheduling, the *A*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *A*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `A(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `A`
Type: character vector
Values: vector
Default: `'A'`

B-matrix(v1,v2,v3) — *B*-matrix of the state-space implementation

B (default) | array

B-matrix of the state-space implementation. In the case of 3-D scheduling, the *B*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *B*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `B(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `B`
Type: character vector
Values: vector
Default: `'B'`

C-matrix(v1,v2,v3) — *C*-matrix of the state-space implementation

C (default) | array

C-matrix of the state-space implementation. In the case of 3-D scheduling, the *C*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *C*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `C(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `C`
Type: character vector
Values: vector
Default: `'C'`

D-matrix(v1,v2,v3) — *D*-matrix of the state-space implementation

D (default) | array

D-matrix of the state-space implementation. In the case of 3-D scheduling, the *D*-matrix should have five dimensions, the last three corresponding to scheduling variables *v*1, *v*2, and *v*3. For example, if the *D*-matrix corresponding to the first entry of *v*1, the first entry of *v*2, and the first entry of *v*3 is the identity matrix, then `D(:,:,1,1,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `D`
Type: character vector
Values: vector
Default: `'D'`

First scheduling variable (v1) breakpoints — Breakpoints for first scheduling variable

v1_vec (default) | vector

Vector of the breakpoints for the first scheduling variable. The length of *v*1 should be same as the size of the third dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v1`
Type: character vector
Values: vector
Default: `'v1_vec'`

Second scheduling variable (v2) breakpoints — Breakpoints for second scheduling variable

v2_vec (default) | vector

Vector of the breakpoints for the second scheduling variable. The length of *v*2 should be same as the size of the fourth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v2`
Type: character vector
Values: vector
Default: `'v2_vec'`

Third scheduling variable (v3) breakpoints — Breakpoints for third scheduling variable

v3_vec (default) | vector

Vector of the breakpoints for the third scheduling variable. The length of *v*3 should be same as the size of the fifth dimension of *A*, *B*, *C*, and *D*.

Programmatic Use
Block Parameter: `breakpoints_v3`
Type: character vector
Values: vector
Default: `'v3_vec'`

Initial state, x_initial — Initial states

`0` (default) | vector

Vector of initial states for the controller, that is, initial values for the state vector, *x*. It should have length equal to the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Poles of A(v)-H(v)*C(v) — Desired poles

`[-5 -2]` (default) | vector

Vector of the desired poles of *A-HC*. Note that the poles are assigned to the same locations for all values of the scheduling parameter *v*. Hence the number of pole locations defined should be equal to the length of the first dimension of the *A*-matrix.

Programmatic Use
Block Parameter: `vec_w`
Type: character vector
Values: vector
Default: `'[-5 -2]'`

Algorithms

The block implements a gain-scheduled state-space controller as defined by the equations:

$$\dot{x} = A(v)x + B(v)y$$
$$u = C(v)x + D(v)y$$

in the self-conditioned form

$$\dot{z} = (A(v) - H(v)C(v))z + (B(v) - H(v)D(v))e + H(v)u_{meas}$$
$$u_{dem} = C(v)z + D(v)e$$

For the rationale behind this self-conditioned implementation, refer to the Self-Conditioned [A,B,C,D] block reference. These blocks implement a gain-scheduled version of the Self-Conditioned [A,B,C,D] block, *v* being the vector of parameters over which *A*, *B*, *C*, and *D* are defined. This type of controller scheduling assumes that the matrices *A*, *B*, *C*, and *D* vary smoothly as a function of *v*, which is often the case in aerospace applications.

Version History

Introduced before R2006a

References

[1] Kautsky, Nichols, and Van Dooren. "Robust Pole Assignment in Linear State Feedback." *International Journal of Control*, Vol. 41, Number 5, 1985, pp. 1129-1155.

Extended Capabilities

C/C++ Code Generation

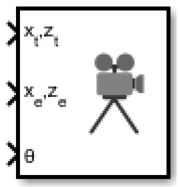
Generate C and C++ code using Simulink® Coder™.

See Also

1D Self-Conditioned [A(v),B(v),C(v),D(v)] | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | 3D Controller [A(v),B(v),C(v),D(v)] | 3D Observer Form [A(v),B(v),C(v),F(v),H(v)] | Linear Second-Order Actuator | Nonlinear Second-Order Actuator

3DoF Animation

Create 3-D MATLAB Graphics animation of three-degrees-of-freedom object



Libraries:
Aerospace Blockset / Animation / MATLAB-Based Animation

Description

The 3DoF Animation block displays a 3-D animated view of a three-degrees-of-freedom (3DoF) craft, its trajectory, and its target using MATLAB Graphics.

The 3DoF Animation block uses input values and dialog parameters to create and display the animation.

This block does not produce deployable code, but you can use it with Simulink Coder external mode as a SimViewingDevice.

Ports

Input

$x_t z_t$ — Target downrange position and altitude (positive down)
two-element vector

Downrange position and altitude (positive down) of the target, specified as a two-element vector.

Data Types: `double`

$x_e z_e$ — Craft downrange position and altitude (positive down)
two-element vector

Downrange position and altitude (positive down) of the craft, specified as a two-element vector.

Data Types: `double`

θ — Attitude of craft
1-by-1 scalar

Attitude of the craft, specified as 1-by-1 scalar, in radians.

Data Types: `double`

Parameters

Axes limits [xmin xmax ymin ymax zmin zmax] — Axes limits

`[0 5000 -2000 2000 -5050 -3050]` (default) | six-element vector

Three-dimensional space to be viewed, specified as a six-element vector.

Programmatic Use
Block Parameter: `u1`
Type: character vector
Values: six-element vector
Default: `'[0 5000 -2000 2000 -5050 -3050]'`

Time interval between updates — Time interval

`0.05` (default) | scalar

Time interval at which the animation is redrawn, specified as a double scalar.

Programmatic Use
Block Parameter: `u2`
Type: character vector
Values: double scalar
Default: `'0.05'`

Size of craft displayed — Scale factor

`1.0` (default) | scalar

Scale factor to adjust the size of the craft and target, specified as a double scalar.

Programmatic Use
Block Parameter: `u3`
Type: character vector
Values: double scalar
Default: `'1.0'`

Enter view — Entrance view

`Fixed position` (default) | `Cockpit` | `Fly alongside`

Preset entrance views, specified as:

- `Fixed position`
- `Cockpit`
- `Fly alongside`

These preset views are specified by MATLAB Graphics parameters **CameraTarget** and **CameraUpVector** for the figure axes.

Tip To customize the position and field of view for the selected view, use the **Position of camera** and **View angle** parameters.

Programmatic Use
Block Parameter: `u5`
Type: character vector

Values: `Fixed position` | `Cockpit` | `Fly alongside`
Default: `'Fixed position'`

Position of camera [xc yc zc] — Camera position

`[2000 500 -3150]` (default) | three-element vector

Camera position, specified using the MATLAB Graphics parameter CameraPosition for the figure axes as a three-element vector. Used in all cases except for the Cockpit view.

Programmatic Use
Block Parameter: `u6`
Type: character vector
Values: three-element vector
Default: `'[2000 500 -3150]'`

View angle — View angle

`10` (default) | scalar

View angle, specified as MATLAB Graphics parameter CameraViewAngle for the figure axes in degrees as a double scalar.

Programmatic Use
Block Parameter: `u7`
Type: character vector
Values: double scalar
Default: `'10'`

Enable animation — Display animation

`on` (default) | `off`

To display the animation during the simulation, select this check box. If not selected, the animation is not displayed.

Programmatic Use
Block Parameter: `u8`
Type: character vector
Values: `on` | `off`
Default: `'on'`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

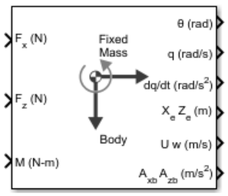
6DoF Animation | FlightGear Preconfigured 6DoF Animation | CameraPosition | CameraViewAngle

Topics

"Design a Guidance System in MATLAB and Simulink"

3DOF (Body Axes)

Implement three-degrees-of-freedom equations of motion with respect to body axes



Libraries:
Aerospace Blockset / Equations of Motion / 3DOF

Description

The 3DOF (Body Axes) block implements three-degrees-of-freedom equations of motion with respect to body axes. It considers the rotation in the vertical plane of a body-fixed coordinate frame about a flat Earth reference frame. For more information about the rotation and equations of motion, see “Algorithms” on page 5-60.

Ports

Input

F_x — Applied force along *x*-axis
scalar

Applied force along the body *x*-axis, specified as a scalar, in the units selected in **Units**.

Data Types: `double`

F_z — Applied force along *z*-axis
scalar

Applied force along the body *z*-axis, specified as a scalar.

Data Types: `double`

M — Applied pitching moment
scalar

Applied pitching moment, specified as a scalar.

Data Types: `double`

g — Gravity
scalar

Gravity, specified as a scalar.

Dependencies

To enable this port, set **Gravity source** to `External`.

Data Types: `double`

Output

θ — Pitch altitude
scalar

Pitch attitude, within ±pi, returned as a scalar, in radians.

Data Types: `double`

q — Pitch angular rate
scalar

Pitch angular rate, returned as a scalar, in radians per second.

Data Types: `double`

dq/dt — Pitch angular acceleration
scalar

Pitch angular acceleration, returned as a scalar, in radians per second squared.

Data Types: `double`

X_eZ_e — Location of body
two-element vector

Location of the body in the flat Earth reference frame, (*Xe, Ze*), returned as a two-element vector.

Data Types: `double`

U w — Velocity of body
two-element vector

Velocity of the body resolved into the body-fixed coordinate frame, (*u, w*), returned as a two-element vector.

Data Types: `double`

$A_{xb}A_{zb}$ — Acceleration of body
two-element vector

Acceleration of the body with respect to the body-fixed coordinate frame, (*Ax, Az*), returned as a two-element vector.

Data Types: `double`

$A_{xe}A_{ze}$ — Acceleration of body
two-element vector

Accelerations of the body with respect to the inertial (flat Earth) coordinate frame, returned as a two-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this port, select the **Include inertial acceleration** check box.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Axes — Body or wind axes

`Body` (default) | `Wind`

Body or wind axes, specified as `Wind` or `Body`

Programmatic Use
Block Parameter: `axes`
Type: character vector
Values: `Wind` | `Body`
Default: `Body`

Mass Type — Mass type
`Fixed` (default) | `Simple Variable` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 3DOF (Body Axes) • 3DOF (Wind Axes)

Mass Type	Description	Default for
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 3DOF (Body Axes) • Simple Variable Mass 3DOF (Wind Axes)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 3DOF (Body Axes) • Custom Variable Mass 3DOF (Wind Axes)

The `Fixed` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Fixed'`

Initial velocity — Initial velocity of body

`100` (default) | scalar

Initial velocity of the body, (V_0), specified as a scalar.

Programmatic Use
Block Parameter: `v_ini`
Type: character vector
Values: `'100'` | scalar
Default: `'100'`

Initial body attitude — Initial pitch altitude

`0` (default) | scalar

Initial pitch attitude of the body, (θ_0), specified as a scalar.

Programmatic Use
Block Parameter: `theta_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial body rotation rate — Initial pitch rotation rate

`0` (default) | scalar

Initial pitch rotation rate, (q_0), specified as a scalar.

Programmatic Use
Block Parameter: `q_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial incidence — Initial angle

`0` (default) | scalar

Initial angle between the velocity vector and the body, (α_0), specified as a scalar.

Programmatic Use
Block Parameter: `alpha_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial position (x,z) — Initial location

`[0 0]` (default) | two-element vector

Initial location of the body in the flat Earth reference frame, specified as a two-element vector.

Programmatic Use
Block Parameter: `pos_ini`
Type: character vector
Values: `'[0 0]'` | two-element vector
Default: `'[0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a scalar.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: `'1.0'` | scalar
Default: `'1.0'`

Inertia — Inertia

`1.0` (default) | scalar

Inertia of the body, specified as a scalar.

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use
Block Parameter: `Iyy`
Type: character vector
Values: `'1.0'` | scalar
Default: `'1.0'`

Gravity Source — Gravity source

`Internal` (default) | `External`

Gravity source, specified as:

External	Variable gravity input to block
Internal	Constant gravity specified in mask

Programmatic Use
Block Parameter: g_in
Type: character vector
Values: 'Internal' | 'External'
Default: 'Internal'

Acceleration due to gravity — Gravity source

9.81 (default) | scalar

Acceleration due to gravity, specified as a double scalar and used if internal gravity source is selected. If gravity is to be neglected in the simulation, this value can be set to 0.

Dependencies

- To enable this parameter, set **Gravity Source** to Internal.

Programmatic Use
Block Parameter: g
Type: character vector
Values: '9.81' | scalar
Default: '9.81'

Include inertial acceleration — Include inertial acceleration port

off (default) | on

Select this check box to add an inertial acceleration in flat Earth frame output port. You typically connect this signal to the accelerometer.

Dependencies

To enable the $\mathbf{A_{xe}A_{ze}}$ port, select this parameter.

Programmatic Use
Block Parameter: abi_flag
Type: character vector
Values: 'off' | 'on'
Default: 'off'

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- The number of names must match the number of states, as shown for each item, or be empty. Set all or none of the block states.
- To assign names to single-variable states, enter unique names between quotes, for example, 'q' or "q".
- To assign names to two-variable states, enter a comma-separated list surrounded by braces, for example, {'Xe','Ze'}.

- If a state parameter is empty (' '), no name is assigned.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array of character vectors, or string.

Velocity: e.g., {'u, 'w'} — Velocity state name

'' (default) | comma-separated list surrounded by braces

Velocity state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `vel_statename`
Type: character vector
Values: '' | comma-separated list surrounded by braces
Default: ''

Position: e.g., {'Xe', 'Ze'} — Position state name

'' (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pos_statename`
Type: character vector
Values: '' | comma-separated list surrounded by braces
Default: ''

Pitch angular rate e.g., 'q' — Pitch angular rate state name

'' (default)

Pitch angular rate state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `q_statename`
Type: character vector | string
Values: '' | scalar
Default: ''

Pitch attitude: e.g., 'theta' — Pitch attitude state name

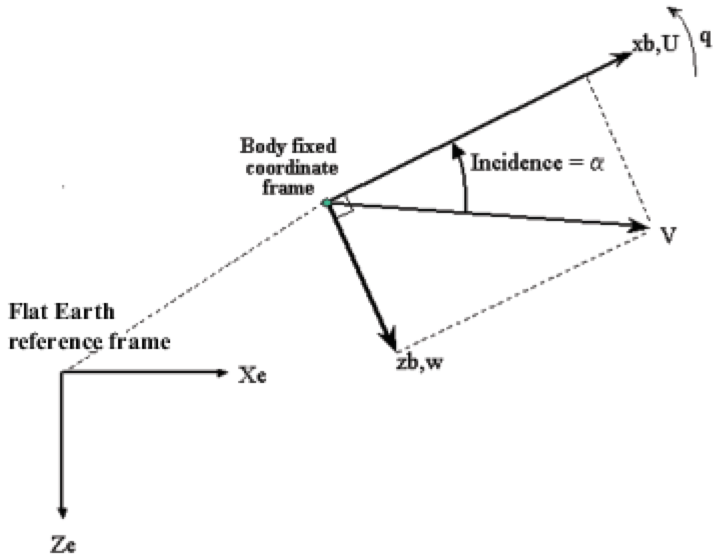
'' (default)

Pitch attitude state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `theta_statename`
Type: character vector | string
Values: ''
Default: ''

Algorithms

The block considers the rotation in the vertical plane of a body-fixed coordinate frame about a flat Earth reference frame.



The equations of motion are

$$A_{xb} = \dot{u} = A_{xe} - qw$$

$$A_{zb} = \dot{w} = A_{ze} + qu$$

$$A_{xe} = \frac{F_x}{m} - g\sin\theta$$

$$A_{ze} = \frac{F_z}{m} + g\cos\theta$$

$$\dot{X}_e = u\cos\theta + w\sin\theta$$

$$\dot{Z}_e = -u\sin\theta + w\cos\theta$$

$$\dot{q} = \frac{M_y}{I_{yy}}$$

$$\dot{\theta} = q$$

where the applied forces are assumed to act at the center of gravity of the body. Input variables are F_x, F_z, M_y. g is an optional input variable.

Version History

Introduced in R2006a

R2021b: 3DOF (Body Axes) Block Changes

Behavior changed in R2021b

The 3DOF equations of motion have been updated. Existing models created prior to R2021b that contain 3DOF equations of motion blocks continue to run. If you replace a pre-R2021b version of a

3DOF equation of motion block with an R2021b or later version, your updated model might have a higher tendency for algebraic loops. For an example of how to remove algebraic loops using unit delays, see “Remove Algebraic Loops”. For further information about algebraic loops, see “Identify Algebraic Loops in Your Model”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

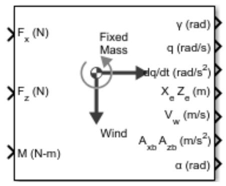
3DOF (Wind Axes) | 4th Order Point Mass (Longitudinal) | Custom Variable Mass 3DOF (Body Axes) | Custom Variable Mass 3DOF (Wind Axes) | Simple Variable Mass 3DOF (Body Axes) | Simple Variable Mass 3DOF (Wind Axes)

Topics

“Design a Guidance System in MATLAB and Simulink”

3DOF (Wind Axes)

Implement three-degrees-of-freedom equations of motion with respect to wind axes



Libraries:
Aerospace Blockset / Equations of Motion / 3DOF

Description

The 3DOF (Wind Axes) block implements three-degrees-of-freedom equations of motion with respect to wind axes. It considers the rotation in the vertical plane of a wind-fixed coordinate frame about a flat Earth reference frame. For more information about the rotation and equations of motion, see “Algorithms” on page 5-69.

Limitations

The block assumes that the applied forces act at the center of gravity of the body, and that the mass and inertia are constant.

Ports

Input

$\mathbf{F_x}$ — Applied force along wind *x*-axis
scalar

Applied force along the wind *x*-axis, specified as a scalar, in the units selected in **Units**.

Data Types: `double`

$\mathbf{F_z}$ — Applied force along wind *z*-axis
scalar

Applied force along the wind *z*-axis, specified as a scalar.

Data Types: `double`

M — Applied pitching moment
scalar

Applied pitching moment, specified as a scalar.

Data Types: `double`

g — Gravity
scalar

Gravity, specified as a scalar.

Dependencies

To enable this port, set **Gravity source** to `External`.

Data Types: `double`

Output

γ — Flight path angle
scalar

Flight path angle, within ±pi, returned as a scalar, in radians.

Data Types: `double`

q — Pitch angular rate
scalar

Pitch angular rate, returned as a scalar, in radians per second.

Data Types: `double`

dq/dt — Pitch angular acceleration
scalar

Pitch angular acceleration, returned as a scalar, in radians per second squared.

Data Types: `double`

X_eZ_e — Location of body
two-element vector

Location of the body in the flat Earth reference frame, (*Xe, Ze*), returned as a two-element vector.

Data Types: `double`

V_w — Velocity in wind-fixed frame
two-element vector

Velocity of the body resolved into the wind-fixed coordinate frame, (*V, 0*), returned as a two-element vector.

Data Types: `double`

$A_{xb}A_{zb}$ — Acceleration of body
two-element vector

Acceleration of the body with respect to the body-fixed coordinate frame, (*Ax, Az*), returned as a two-element vector.

Data Types: `double`

α — Angle of attack
scalar

Angle of attack, returned as a scalar, in radians.

Data Types: `double`

$A_{xe}A_{ze}$ — Acceleration of body
two-element vector

Accelerations of the body with respect to the inertial (flat Earth) coordinate frame, returned as a two-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this port, select the **Include inertial acceleration** check box.

Data Types: double

Parameters

Main

Units — Input and output units

Metric (MKS) (default) | English (Velocity in ft/s) | English (Velocity in kts)

Input and output units, specified as Metric (MKS), English (Velocity in ft/s), or English (Velocity in kts).

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
Metric (MKS)	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
English (Velocity in ft/s)	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
English (Velocity in kts)	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: units
Type: character vector
Values: Metric (MKS) | English (Velocity in ft/s) | English (Velocity in kts)
Default: Metric (MKS)

Axes — Body or wind axes

Wind (default) | Body

Body or wind axes, specified as Wind or Body

Programmatic Use
Block Parameter: axes
Type: character vector
Values: Wind | Body
Default: Wind

Mass type — Mass type

Fixed (default) | Simple Variable | Custom Variable

Mass type, specified according to the following table.

Mass Type	Description	Default for
Fixed	Mass is constant throughout the simulation.	• 3DOF (Body Axes) • 3DOF (Wind Axes)
Simple Variable	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 3DOF (Body Axes) • Simple Variable Mass 3DOF (Wind Axes)
Custom Variable	Mass and inertia variations are customizable.	• Custom Variable Mass 3DOF (Body Axes) • Custom Variable Mass 3DOF (Wind Axes)

The Fixed selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: mtype
Type: character vector
Values: Fixed | Simple Variable | Custom Variable
Default: 'Fixed'

Initial airspeed — Initial speed

100 (default) | scalar greater than 0

Initial speed of the body, (V_0), specified as a scalar greater than 0.

Programmatic Use
Block Parameter: V_ini
Type: character vector
Values: '100' | scalar
Default: '100'

Initial flight path angle — Initial flight path angle

0 (default) | scalar

Initial flight path angle of the body, (γ_0), specified as a scalar.

Programmatic Use
Block Parameter: gamma_ini
Type: character vector
Values: '0' | scalar
Default: '0'

Initial body rotation rate — Initial pitch rotation rate

0 (default) | scalar

Initial pitch rotation rate, (q_0), specified as a scalar.

Programmatic Use
Block Parameter: `q_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial incidence — Initial angle

`0` (default) | scalar

Initial angle between the velocity vector and the body, (α_0), specified as a scalar.

Programmatic Use
Block Parameter: `alpha_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial position (x,z) — Initial location

`[0 0]` (default) | two-element vector

Initial location of the body in the flat Earth reference frame, specified as a two-element vector.

Programmatic Use
Block Parameter: `pos_ini`
Type: character vector
Values: `'[0 0]'` | two-element vector
Default: `'[0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a scalar.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: `'1.0'` | scalar
Default: `'1.0'`

Inertia body axes — Inertia of body

`1.0` (default) | scalar

Inertia of the body, specified as a scalar.

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use
Block Parameter: `Iyy`
Type: character vector
Values: `'1.0'` | scalar

Default: `'1.0'`

Gravity Source — Gravity source

`Internal` (default) | `External`

Gravity source, specified as:

`External`	Variable gravity input to block
`Internal`	Constant gravity specified in mask

Programmatic Use
Block Parameter: `g_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Acceleration due to gravity — Gravity source

`9.81` (default) | scalar

Acceleration due to gravity, specified as a double scalar and used if internal gravity source is selected. If gravity is to be neglected in the simulation, this value can be set to `0`.

Dependencies

- To enable this parameter, set **Gravity Source** to `Internal`.

Programmatic Use
Block Parameter: `g`
Type: character vector
Values: `'9.81'` | scalar
Default: `'9.81'`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration in flat Earth frame output port. You typically connect this signal to the accelerometer.

Dependencies

To enable the $\mathbf{A_{xe}A_{ze}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- The number of names must match the number of states, as shown for each item, or be empty. Set all or none of the block states.
- To assign names to single-variable states, enter unique names between quotes, for example, `'q'` or `"q"`.
- To assign names to two-variable states, enter a comma-separated list surrounded by braces, for example, `{'Xe','Ze'}`.
- If a state parameter is empty (`' '`), no name is assigned.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array of character vectors, or string.

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `V_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Position: e.g., {'Xe', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pos_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rate: e.g., 'q' — Body rotation state name

`''` (default) | scalar

Body rotation rate state names, specified as a character vector or string.

Programmatic Use
Block Parameter: `q_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Flight path angle: e.g., 'gamma' — Flight path angle state name

`''` (default)

Flight path angle state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `gamma_statename`

Type: character vector | string
Values: `''` | scalar
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

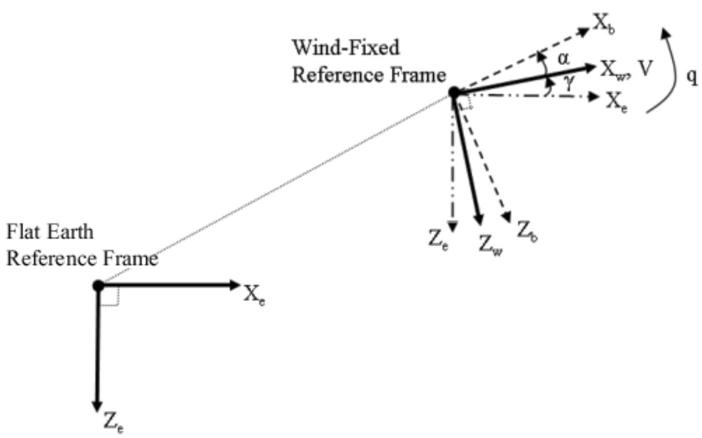
`''` (default) | scalar

Incidence angle state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Algorithms

The block considers the rotation in the vertical plane of a wind-fixed coordinate frame about a flat Earth reference frame.



The equations of motion are

$$A_{xb} = A_{xe} - qV\sin\alpha$$
$$A_{zb} = A_{ze} + qV\cos\alpha$$
$$A_{xe} = \left(\frac{F_x}{m} - g\sin\gamma\right)\cos\alpha - \left(\frac{F_z}{m} + g\cos\gamma\right)\sin\alpha$$
$$A_{ze} = \left(\frac{F_x}{m} - g\sin\gamma\right)\sin\alpha + \left(\frac{F_z}{m} + g\cos\gamma\right)\cos\alpha$$
$$\dot{V} = \frac{F_x}{m} - g\sin\gamma$$
$$\dot{X}_e = V\cos\gamma$$
$$\dot{Z}_e = -V\sin\gamma$$
$$\dot{q} = \frac{M_y}{I_{yy}}$$
$$\dot{\gamma} = q - \dot{\alpha}$$
$$\dot{\alpha} = \frac{F_z}{mV} + \frac{g}{V}\cos\gamma + q$$

where the applied forces are assumed to act at the center of gravity of the body. Input variables are wind-axes forces F_x and F_z and body moment M_y. g is an optional input variable.

Version History

Introduced in R2006a

R2021b: 3DOF (Wind Axes) Block Changes

Behavior changed in R2021b

The 3DOF equations of motion have been updated. Existing models created prior to R2021b that contain 3DOF equations of motion blocks continue to run. If you replace a pre-R2021b version of a 3DOF equation of motion block with an R2021b or later version, your updated model might have a higher tendency for algebraic loops. For an example of how to remove algebraic loops using unit delays, see “Remove Algebraic Loops”. For further information about algebraic loops, see “Identify Algebraic Loops in Your Model”.

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. New York: John Wiley & Sons, 1992.

Extended Capabilities

C/C++ Code Generation

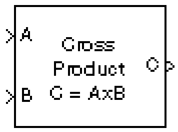
Generate C and C++ code using Simulink® Coder™.

See Also

3DOF (Body Axes) | 4th Order Point Mass (Longitudinal) | Custom Variable Mass 3DOF (Body Axes) | Custom Variable Mass 3DOF (Wind Axes) | Simple Variable Mass 3DOF (Body Axes) | Simple Variable Mass 3DOF (Wind Axes)

3x3 Cross Product

Calculate cross product of two 3-by-1 vectors



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The 3x3 Cross Product block computes cross (or vector) product of two vectors, A and B. The block generates a third vector, C, in a direction normal to the plane containing A and B, with magnitude equal to the product of the lengths of A and B multiplied by the sine of the angle between them. The direction of C follows the right-hand rule in turning from A to B. For related equations, see "Algorithms" on page 5-72.

Ports

Input

A — First cross product input
3-by-1 vector

First cross product input, specified as a vector.

Example: `[10 2 3]`

Data Types: `double`

B — Second cross product input
3-by-1 vector

Second cross product input, specified as a vector.

Example: `[10 2 3]`

Data Types: `double`

Output

C — Cross product
3-by-1 vector

Cross product, output as a vector.

Data Types: `double`

Algorithms

The equations used to calculate A, B, and C are:

$$A = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$$
$$B = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$$
$$C = A \times B = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$
$$= (a_2b_3 - a_3b_2)\mathbf{i} + (a_3b_1 - a_1b_3)\mathbf{j} + (a_1b_2 - a_2b_1)\mathbf{k}$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

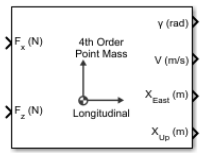
Generate C and C++ code using Simulink® Coder™.

See Also

Create 3x3 Matrix | Adjoint of 3x3 Matrix | Determinant of 3x3 Matrix | Invert 3x3 Matrix

4th Order Point Mass (Longitudinal)

Calculate fourth-order point mass



Libraries:
Aerospace Blockset / Equations of Motion / Point Mass

Description

The 4th Order Point Mass (Longitudinal) block performs the calculations for the translational motion of a single point mass or multiple point masses. For more information on the system for the translational motion of a single point mass or multiple mass, see “Algorithms” on page 5-77.

The 4th Order Point Mass (Longitudinal) block port labels change based on the input and output units selected from the **Units** list.

Limitations

The flat Earth reference frame is considered inertial, an approximation that allows the forces due to the Earth's motion relative to the “fixed stars” to be neglected.

Ports

Input

Port_1 — Force in *x*-axis
scalar | array

Force in *x*-axis, specified as a scalar or array, in selected units.

Data Types: `double`

Port_2 — Force in *z*-axis
scalar | array

Force in *z*-axis, specified as a scalar or array, in selected units.

Data Types: `double`

Output

Port_1 — Flight path angle
scalar | array

Flight path angle, returned as a scalar or array, in radians.

Data Types: `double`

Port_2 — Airspeed
scalar | array

Airspeed, returned as a scalar or array, in selected units.

Data Types: `double`

Port_3 — Downrange or amount traveled east
scalar | array

Downrange or amount traveled east, returned as a scalar or array, in selected units.

Data Types: `double`

Port_4 — Altitude or amount traveled up
scalar | array

Altitude or amount traveled up, returned as a scalar or array, in selected units.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Forces	Velocity	Position	Mass
`Metric (MKS)`	newtons	meters per second	meters	kilograms
`English (Velocity in ft/s)`	pounds	feet per second	feet	slugs
`English (Velocity in kts)`	pounds	knots	feet	slugs

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Initial flight path angle — Initial flight path angle

`0` (default) | scalar | vector

Initial flight path angle of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `gamma0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial airspeed — Initial airspeed

`100` (default) | scalar | vector

Initial airspeed of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `V0`
Type: character vector
Values: scalar | vector
Default: `'100'`

Initial downrange [East] — Initial downrange

`0` (default) | scalar | vector

Initial downrange of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `x0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial altitude [Up] — Initial altitude of point masses

`0` (default) | scalar | vector

Initial altitude of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `h0`
Type: character vector
Values: scalar | vector
Default: `'0'`

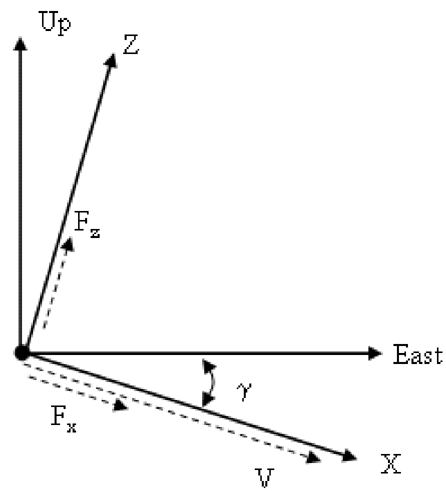
Initial mass — Point mass

`1.0` (default) | scalar | vector

Mass of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `mass0`
Type: character vector
Values: scalar | vector
Default: `'1.0'`

Algorithms



The translational motions of the point mass $[X_{\mathrm{East}} X_{\mathrm{Up}}]^{\mathrm{T}}$ are functions of airspeed (V) and flight path angle (γ),

$$F_x = m\dot{V}$$
$$F_z = mV\dot{\gamma}$$
$$\dot{X}_{East} = V\cos\gamma$$
$$\dot{X}_{Up} = V\sin\gamma$$

where the applied forces $[F_x F_z]^{\mathrm{T}}$ are in a system defined as follows: x-axis is in the direction of vehicle velocity relative to air, z-axis is upward, and y-axis completes the right-handed frame. The mass of the body m is assumed constant.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

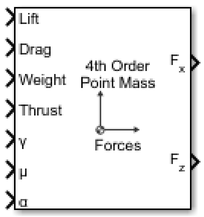
Generate C and C++ code using Simulink® Coder™.

See Also

Simple Variable Mass 3DOF (Body Axes) | Custom Variable Mass 3DOF (Wind Axes) | 4th Order Point Mass Forces (Longitudinal) | 3DOF (Body Axes) | 3DOF (Wind Axes) | 6th Order Point Mass (Coordinated Flight) | Custom Variable Mass 3DOF (Body Axes) | 6th Order Point Mass Forces (Coordinated Flight) | Simple Variable Mass 3DOF (Wind Axes)

4th Order Point Mass Forces (Longitudinal)

Calculate forces used by fourth-order point mass



Libraries:
Aerospace Blockset / Equations of Motion / Point Mass

Description

The 4th Order Point Mass Forces (Longitudinal) block calculates the applied forces for a single point mass or multiple point masses. For more information on the system for the applied forces, see "Algorithms" on page 5-80.

Limitations

The flat Earth reference frame is considered inertial, an approximation that allows the forces due to the Earth motion relative to the "fixed stars" to be neglected.

Ports

Input

Lift — Lift
scalar | array

Lift, specified as a scalar or array, in units of force.

Data Types: `double`

Drag — Drag
scalar | array

Drag, specified as a scalar or array, in units of force.

Data Types: `double`

Weight — Weight
scalar | array

Weight, specified as a scalar or array, in units of force.

Data Types: `double`

Thrust — Thrust
scalar | array

Thrust, specified as a scalar or array, in units of force.

Data Types: `double`

γ — Flight path angle
scalar | array

Flight path angle, specified as a scalar or array, in radians.

Data Types: `double`

μ — Bank angle
scalar | array

Bank angle, specified as a scalar or array, in radians.

Data Types: `double`

α — Angle of attack
scalar | array

Angle of attack, specified as a scalar or array, in radians.

Data Types: `double`

Output

F_x — Force in *x*-axis
scalar | array

Force in *x*-axis, returned as a scalar or array, in units of force.

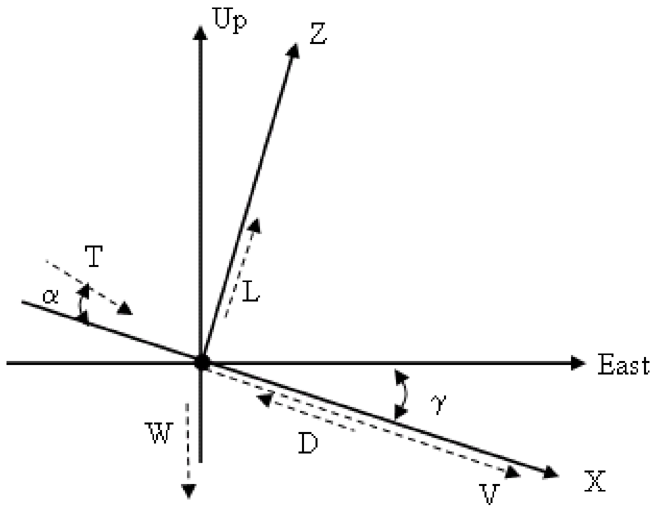
Data Types: `double`

F_z — Force in *z*-axis
scalar | array

Force in *z*-axis, returned as a scalar or array, in units of force.

Data Types: `double`

Algorithms



The applied forces $[F_x\ F_z]^T$ are in a system defined as follows: *x*-axis is in the direction of vehicle velocity relative to air, *z*-axis is upward, and *y*-axis completes the right-handed frame. They are functions of lift (*L*), drag (*D*), thrust (*T*), weight (*W*), flight path angle (*γ*), angle of attack (*α*), and bank angle (*μ*).

$$F_z = (L + T\sin\alpha)\cos\mu - W\cos\gamma$$

$$F_x = T\cos\alpha - D - W\sin\gamma$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

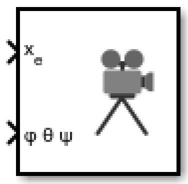
Generate C and C++ code using Simulink® Coder™.

See Also

6th Order Point Mass (Coordinated Flight) | 4th Order Point Mass (Longitudinal) | 6th Order Point Mass Forces (Coordinated Flight)

6DoF Animation

Create 3-D MATLAB Graphics animation of six-degrees-of-freedom object



Libraries:
Aerospace Blockset / Animation / MATLAB-Based Animation

Description

The 6DoF Animation block displays a 3-D animated view of a six-degrees-of-freedom (6DoF) vehicle, its trajectory, and its target using MATLAB Graphics.

The 6DoF Animation block uses the input values and the block parameters to create and display the animation. The **Axes limits**, **Static object position**, and **Position of camera** parameters have the same units of length as the input parameters.

This block does not produce deployable code, but you can use it with Simulink Coder external mode as a SimViewingDevice.

Ports

Input

x_e — Downrange position, crossrange position, and altitude (positive down)
three-element vector

Downrange position, crossrange position, and altitude (positive down) of the vehicle, specified as a three-element vector.

Data Types: `double`

φ θ ψ — Euler angles
three-element vector

Euler angles of the vehicle, specified as a three-element vector.

Data Types: `double`

Parameters

Axes limits [xmin xmax ymin ymax zmin zmax] — Axes limits

`[0 4000 -2000 2000 -5000 -3000]` (default) | six-element vector

Three-dimensional space to be viewed, specified as a six-element vector.

Programmatic Use
Block Parameter: `u1`
Type: character vector
Values: six-element vector
Default: `'[0 4000 -2000 2000 -5000 -3000]'`

Time interval between updates — Time interval

`0.1` (default) | scalar

Time interval at which the animation is redrawn, specified as a double scalar.

Programmatic Use
Block Parameter: `u2`
Type: character vector
Values: double scalar
Default: `'0.1'`

Size of craft displayed — Scale factor

`1.0` (default) | scalar

Scale factor to adjust the size of the vehicle and target, specified as a double scalar.

Programmatic Use
Block Parameter: `u3`
Type: character vector
Values: double scalar
Default: `'1.0'`

Static object position [xp yp zp] — Static object position

`[4000 0 -5000]` (default) | three-element vector

Altitude, crossrange position, and downrange position of the target, specified as three-element vector.

Programmatic Use
Block Parameter: `u4`
Type: character vector
Values: three-element vector
Default: `'[4000 0 -5000]'`

Enter view — Entrance view

`Fixed position` (default) | `Cockpit` | `Fly alongside`

Preset entrance views, specified as:

- `Fixed position`
- `Cockpit`
- `Fly alongside`

These preset views are specified by MATLAB Graphics parameters **CameraTarget** and **CameraUpVector** for the figure axes.

Tip To customize the position and field of view for the selected view, use the **Position of camera** and **View angle** parameters.

Programmatic Use
Block Parameter: `u5`
Type: character vector
Values: `Fixed position` | `Cockpit` | `Fly alongside`
Default: `'Fixed position'`

Position of camera [xc yc zc] — Camera position

`[2000 500 -3150]` (default) | three-element vector

Camera position, specified using the MATLAB Graphics parameter CameraPosition for the figure axes as a three-element vector. Used in all cases except for when **Enter view** is set to `Cockpit`.

Programmatic Use
Block Parameter: `u6`
Type: character vector
Values: three-element vector
Default: `'[2000 500 -3150]'`

View angle — View angle

`10` (default) | scalar

View angle for the MATLAB Graphics parameter CameraViewAngle for the figure axes in degrees, specified as a double scalar.

Programmatic Use
Block Parameter: `u7`
Type: character vector
Values: double scalar
Default: `'10'`

Enable animation — Display animation

`on` (default) | `off`

Whether to display the animation during the simulation. If not selected, the animation is not displayed.

Programmatic Use
Block Parameter: `u8`
Type: character vector
Values: `on` | `off`
Default: `'on'`

Version History

Introduced before R2006a

Extended Capabilities

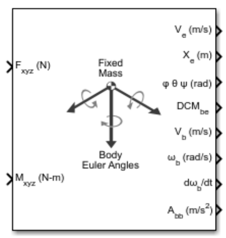
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

3DoF Animation | FlightGear Preconfigured 6DoF Animation | CameraPosition | CameraViewAngle

6DOF (Euler Angles)

Implement Euler angle representation of six-degrees-of-freedom equations of motion



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The 6DOF (Euler Angles) block implements the Euler angle representation of six-degrees-of-freedom equations of motion, taking into consideration the rotation of a body-fixed coordinate frame (X_b, Y_b, Z_b) about a flat Earth reference frame (X_e, Y_e, Z_e). For more information about these reference points, see “Algorithms” on page 5-91.

Limitations

The block assumes that the applied forces act at the center of gravity of the body, and that the mass and inertia are constant.

Ports

Input

F_{xyz}(N) — Applied forces
three-element vector

Applied forces, specified as a three-element vector in body-fixed axes. For more information on the frame, see “Body Coordinates” on page 2-9.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector in body-fixed axes. For more information on the frame, see “Body Coordinates” on page 2-9.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

φ θ ψ (rad) — Euler rotation angles
three-element vector

Euler rotation angles [roll, pitch, yaw] defining an intrinsic *x*-*y*-*z* rotation, as a three-element vector, in radians. Yaw, pitch, and roll angles are applied using the *z*-*y*-*x* rotation sequence, such as `angle2dcm(yaw,pitch,roll,"ZYX")`.

Data Types: `double`

DCM_{be} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity in the body-fixed frame
three-element vector

Velocity in the body-fixed frame, returned as a three-element vector.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)

Mass Type	Description	Default for
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Euler Angles` (default) | `Quaternion`

Equations of motion representation, specified according to the following table.

Representation	Description
`Euler Angles`	Use Euler angles within equations of motion.
`Quaternion`	Use quaternions within equations of motion.

The `Quaternion` selection conforms the equations of motion in “Algorithms” on page 5-91.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Euler Angles` | `Quaternion`
Default: `'Euler Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial velocity in body axes [U,v,w] — Velocity in body axes

`[0 0 0]` (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

`[0 0 0]` (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`

Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Inertia — Inertia

`eye(3)` (default) | scalar

Inertia of the body, specified as a double scalar.

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use
Block Parameter: `inertia`
Type: character vector
Values: `eye(3)` | double scalar
Default: `eye(3)`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name assignment occurs.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

'' (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

'' (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Euler rotation angles: e.g., {'phi', 'theta', 'psi'} — Euler rotation state name

'' (default) | comma-separated list surrounded by braces

Euler rotation angle state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `eul_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

'' (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

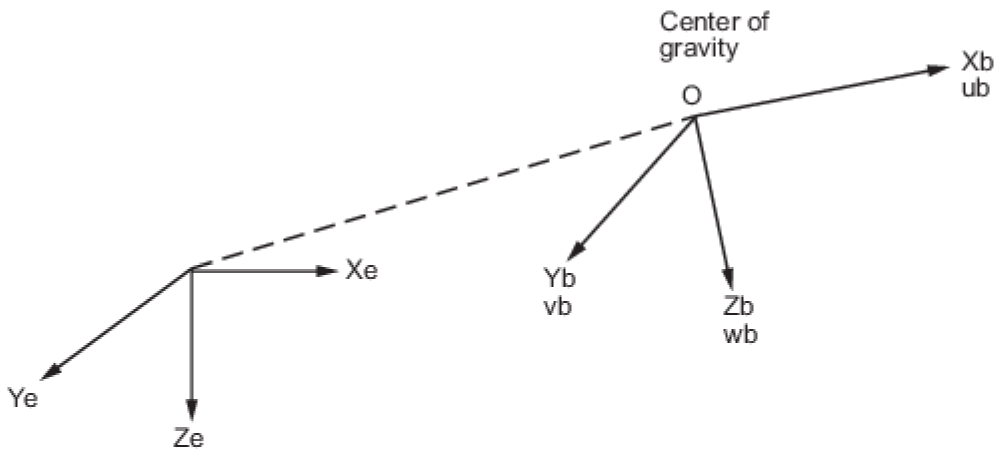
Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Algorithms

The 6DOF (Euler Angles) block uses these reference frame concepts.

- The origin of the body-fixed coordinate frame is the center of gravity of the body, and the body is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass.

The flat Earth reference frame is considered inertial, an excellent approximation that allows the forces due to the Earth motion relative to the "fixed stars" to be neglected.



Flat Earth reference frame

- Translational motion of the body-fixed coordinate frame, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the body-fixed frame, and the mass of the body m is assumed constant.

$$\bar{F}_b = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m\left(\dot{\bar{V}}_b + \bar{\omega} \times \bar{V}_b\right)$$

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{w}_b \end{bmatrix} = \frac{1}{m}\bar{F}_b - \bar{\omega} \times \bar{V}_b$$

$$A_{be} = \frac{1}{m}F_b$$

$$\bar{V}_b = \begin{bmatrix} u_b \\ v_b \\ w_b \end{bmatrix}, \bar{\omega} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

- The rotational dynamics of the body-fixed frame, where the applied moments are $[L\ M\ N]^T$, and the inertia tensor I is with respect to the origin O.

$$\bar{M}_B = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\bar{\omega}} + \bar{\omega} \times (I\bar{\omega})$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

- The relationship between the body-fixed angular velocity vector, $[p\ q\ r]^T$, and the rate of change of the Euler angles, $\left[\dot{\phi}\ \ \dot{\theta}\ \ \dot{\psi}\right]^T$, are determined by resolving the Euler rates into the body-fixed coordinate frame.

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \equiv J^{-1} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

Inverting J then gives the required relationship to determine the Euler rate vector.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = J \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & (\sin\phi\tan\theta) & (\cos\phi\tan\theta) \\ 0 & \cos\phi & -\sin\phi \\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis, *Aircraft Control and Simulation*. Hoboken, NJ: Second Edition, John Wiley & Sons, 2003.

[2] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. Reston, Va: Second Edition, AIAA Education Series, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

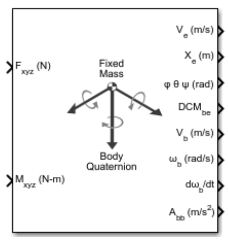
6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Topics

“About Aerospace Coordinate Systems” on page 2-8

6DOF (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion with respect to body axes



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The 6DOF (Quaternion) block implements quaternion representation of six-degrees-of-freedom equations of motion with respect to body axes. For a description of the coordinate system and the translational dynamics, see the block description for the 6DOF (Euler Angles) block.

For more information on the integration of the rate of change of the quaternion vector, see "Algorithms" on page 5-101.

Limitations

The block assumes that the applied forces act at the center of gravity of the body, and that the mass and inertia are constant.

Ports

Input

F_{xyz}(N) — Applied forces
three-element vector

Applied forces, specified as a three-element vector in body-fixed axes. For more information on the frame, see "Body Coordinates" on page 2-9.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector in body-fixed axes. For more information on the frame, see "Body Coordinates" on page 2-9.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

φ θ ψ (rad) — Euler rotation angles
three-element vector

Euler rotation angles [roll, pitch, yaw] defining an intrinsic *x*-*y*-*z* rotation, as a three-element vector, in radians. Yaw, pitch, and roll angles are applied using the *z*-*y*-*x* rotation sequence, such as `angle2dcm(yaw,pitch,roll,"ZYX")`.

Data Types: `double`

DCM_{be} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity in the body-fixed frame
three-element vector

Velocity in the body-fixed frame, returned as a three-element vector.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use

Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)

Mass Type	Description	Default for
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Quaternion` (default) | `Euler Angles`

Equations of motion representation, specified according to the following table.

Representation	Description
`Euler Angles`	Use Euler angles within equations of motion.
`Quaternion`	Use quaternions within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-101.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Euler Angles` | `Quaternion`
Default: `'Quaternion'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

[0 0 0] (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: xme_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial velocity in body axes [U,v,w] — Velocity in body axes

[0 0 0] (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: Vm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

[0 0 0] (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: eul_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial body rotation rates [p,q,r] — Initial body rotation

[0 0 0] (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: pm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial mass — Initial mass

1.0 (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: mass_0

Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Inertia — Inertia

`eye(3)` (default) | scalar

Inertia of the body, specified as a double scalar.

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use
Block Parameter: `inertia`
Type: character vector
Values: `eye(3)` | double scalar
Default: `eye(3)`

Gain for quaternion normalization — Gain

`1.0` (default) | scalar

Gain to maintain the norm of the quaternion vector equal to 1.0, specified as a double scalar.

Programmatic Use
Block Parameter: `k_quat`
Type: character vector
Values: `1.0` | double scalar
Default: `1.0`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.

- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`

Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Algorithms

The integration of the rate of change of the quaternion vector is given below. The gain K drives the norm of the quaternion state vector to 1.0 should εbecome nonzero. You must choose the value of this gain with care, because a large value improves the decay rate of the error in the norm, but also slows the simulation because fast dynamics are introduced. An error in the magnitude in one element of the quaternion vector is spread equally among all the elements, potentially increasing the error in the state vector.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = 1/2 \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} + K\varepsilon \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

$$\varepsilon = 1 - \left(q_0^2 + q_1^2 + q_2^2 + q_3^2\right)$$

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Version History

Introduced in R2006a

Extended Capabilities

C/C++ Code Generation

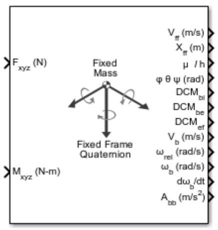
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

6DOF ECEF (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion in Earth-centered Earth-fixed (ECEF) coordinates



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The 6DOF ECEF (Quaternion) block Implement quaternion representation of six-degrees-of-freedom equations of motion in Earth-centered Earth-fixed (ECEF) coordinates. It considers the rotation of a Earth-centered Earth-fixed (ECEF) coordinate frame (X_{ECEF}, Y_{ECEF}, Z_{ECEF}) about an Earth-centered inertial (ECI) reference frame (X_{ECI}, Y_{ECI}, Z_{ECI}). The origin of the ECEF coordinate frame is the center of the Earth. For more information on the ECEF coordinate frame, see “Algorithms” on page 5-113.

Limitations

- This implementation assumes that the applied forces act at the center of gravity of the body, and that the mass and inertia are constant.
- This implementation generates a geodetic latitude that lies between ±90 degrees, and longitude that lies between ±180 degrees. Additionally, the MSL altitude is approximate.
- The Earth is assumed to be ellipsoidal. By setting flattening to 0.0, a spherical planet can be achieved. The Earth's precession, nutation, and polar motion are neglected. The celestial longitude of Greenwich is Greenwich Mean Sidereal Time (GMST) and provides a rough approximation to the sidereal time.
- The implementation of the ECEF coordinate system assumes that the origin is at the center of the planet, the *x*-axis intersects the Greenwich meridian and the equator, the *z*-axis is the mean spin axis of the planet, positive to the north, and the *y*-axis completes the right-handed system.
- The implementation of the ECI coordinate system assumes that the origin is at the center of the planet, the *x*-axis is the continuation of the line from the center of the Earth toward the vernal equinox, the *z*-axis points in the direction of the mean equatorial plane's north pole, positive to the north, and the *y*-axis completes the right-handed system.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

$L_G(0)$ — Initial celestial longitude of Greenwich
scalar

Greenwich meridian initial celestial longitude angle, specified as a scalar.

Dependencies

To enable this port

- Set **Celestial longitude of Greenwich** to `External`.
- set **Planet model** to `Earth`.

Data Types: `double`

$L_{PM}(0)$ — Prime meridian initial celestial longitude angle
scalar

Prime meridian initial celestial longitude angle, specified as a scalar.

Dependencies

To enable this port

- Set **Celestial longitude of prime meridian** to `External`.
- Set **Planet model** to `Custom`.

Data Types: `double`

Output

V_{ff} — Velocity of body with respect to ECEF frame,
three-element vector

Velocity of body with respect to ECEF frame, expressed in ECEF frame, returned as a three-element vector.

Data Types: `double`

X_{ff} — Position in ECEF reference frame
three-element vector

Position in ECEF reference frame, returned as a three-element vector.

Data Types: `double`

μ l h — Position in geodetic latitude, longitude, and altitude
three-element vector | M-by-3 array

Position in geodetic latitude, longitude, and altitude, in degrees, returned as a three-element vector or M-by-3 array, in selected units of length, respectively.

Data Types: `double`

φ θ Ψ (rad) — Body rotation angles
three-element vector

Body rotation angles [roll, pitch, yaw], returned as a three-element vector, in radians. Euler rotation angles are those between body and NED coordinate systems.

Data Types: `double`

DCM_{bi} — Coordinate transformation from ECI axes
3-by-3 matrix

Coordinate transformation from ECI axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

DCM_{be} — Coordinate transformation from NED axes to body-fixed axes
3-by-3 matrix

Coordinate transformation from NED axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

DCM_{ef} — Coordinate transformation from fixed-frame axes
3-by-3 matrix

Coordinate transformation from fixed-frame axes to NED axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity of body with respect to fixed-frame
three-element vector

Velocity of body with respect to fixed-frame, returned as a three-element vector.

Data Types: `double`

ω_{rel} — Relative angular rates of body with respect to NED frame
three-element vector

Relative angular rates of body with respect to NED frame, expressed in body frame and returned as a three-element vector, in radians per second.

Data Types: `double`

ω_b — Angular rates of body with respect to inertial frame
three-element vector

Angular rates of the body with respect to inertial frame, expressed in body frame and returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations of the body with respect to inertial frame
three-element vector

Angular accelerations of the body with respect to inertial frame, expressed in body frame and returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations of the body with respect to the fixed-frame, returned as a three-element vector.

Data Types: `double`

$A_{b\ ff}$ — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to fixed-frame, returned as a three-element vector.

Dependencies

To enable this point, **Include inertial acceleration**.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass type — Mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Select the type of mass to use:

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Fixed` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Simple Variable'`

Initial position in geodetic latitude, longitude and altitude [mu,l,h] — Initial location of rigid body

`[0 0 0]` (default) | three-element vector

Initial location of the rigid body in the geodetic reference frame, specified as a three-element vector. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Programmatic Use
Block Parameter: `xg_0`

Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial velocity in body axes [U,v,w] — Velocity in body axes

[0 0 0] (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: Vm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

[0 0 0] (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: eul_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial body rotation rates [p,q,r] — Initial body rotation

[0 0 0] (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: pm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial mass — Initial mass

1.0 (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: mass_0
Type: character vector
Values: '1.0' | double scalar
Default: '1.0'

Inertia — Inertia

`eye(3)` (default) | scalar

Inertia of the body, specified as a double scalar.

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use
Block Parameter: `inertia`
Type: character vector
Values: `eye(3)` | double scalar
Default: `eye(3)`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

Planet

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Equatorial radius of planet — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator, specified as a double scalar, in the same units as the desired units for the ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `R`

Type: character vector
Values: double scalar
Default: '6378137'

Flattening — Flattening of planet

1/298.257223563 (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to Custom.

Programmatic Use
Block Parameter: F
Type: character vector
Values: double scalar
Default: '1/298.257223563'

Rotational rate — Rotational rate

7292115e-11 (default) | scalar

Rotational rate of the planet, specified as a scalar, in rad/s.

Dependencies

To enable this parameter, set **Planet model** to Custom.

Programmatic Use
Block Parameter: w_E
Type: character vector
Values: double scalar
Default: '7292115e-11'

Celestial longitude of Greenwich source — Source of Greenwich meridian initial celestial longitude

Internal (default) | External

Source of Greenwich meridian initial celestial longitude, specified as:

Internal	Use celestial longitude value from **Celestial longitude of Greenwich**.
External	Use external input for celestial longitude value.

Dependencies

- To enable this parameter, set **Planet model** to Earth.
- Setting this parameter to External enables the $\mathbf{L_G(0)}$ port.
- If **Planet model** is set to Custom, the parameter name changes to **Celestial longitude of prime meridian source**.

Programmatic Use
Block Parameter: angle_in

Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Celestial longitude of Greenwich [deg] — Initial angle

`0` (default) | scalar

Initial angle between Greenwich meridian and the *x*-axis of the inertial frame, specified as a double scalar.

Dependencies

- To enable this parameter, set:
  - **Celestial longitude of Greenwich source** to `Internal`.
  - **Planet model** to `Earth (WGS84)`.
- If **Planet model** is set to `Custom`, the parameter name changes to **Celestial longitude of prime meridian [deg]**.

Programmatic Use
Block Parameter: `LPM0`
Type: character vector
Values: double scalar
Default: `'0'`

Celestial longitude of prime meridian source — Source of prime meridian initial celestial longitude
`Internal` (default) | `External`

Source of prime meridian initial celestial longitude, specified as:

`Internal`	Use celestial longitude value from **Celestial longitude of prime meridian**.
`External`	Use external input for celestial longitude value.

Dependencies

- To enable this parameter, set **Planet model** to `Custom`.
- Setting this parameter to `External` enables the $\mathbf{L_{PM}(0)}$ port.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich source**.

Programmatic Use
Block Parameter: `angle_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Celestial longitude of prime meridian [deg] — Initial angle
`0` (default) | scalar

Initial angle between prime meridian and the *x*-axis of the ECI frame, specified as a double scalar.

Dependencies

- To enable this parameter, set:
  - **Celestial longitude of prime meridian source** to `Internal`.
  - **Planet model** to `Custom`.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich [deg]**.

Programmatic Use

Block Parameter: `LPM0`
Type: character vector
Values: double scalar
Default: `'0'`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`' '` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use

Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`' '` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Fixed frame position: e.g., {'Xff', 'Yff', 'Zff'} — ECEF position state name

`''` (default) | comma-separated list surrounded by braces

ECEF position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `posfixedframe_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Inertial position: e.g., {'Xinertial', 'Yinertial', 'Zinertial'} — Inertial position state names

`''` (default) | comma-separated list surrounded by braces

Inertial position state names, specified as a comma-separated list surrounded by braces.

Default value is `''`.

Programmatic Use
Block Parameter: `posinertial_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Celestial longitude of Greenwich: e.g., 'LG' — Celestial longitude state name

`''` (default) | character vector

Celestial longitude of Greenwich state name, specified as a character vector.

Dependencies

- To enable this parameter, set:**Planet model** to `Earth (WGS84)`.
- If **Planet model** is set to `Custom`, the parameter name changes to **Celestial longitude of prime meridian: e.g., 'LPM'**.

Programmatic Use
Block Parameter: `LPM_statename`
Type: character vector
Values: `''` | scalar
Default: `''`

Celestial longitude of prime meridian: e.g., 'LPM' — Celestial longitude prime meridian
`''` (default) | character vector

Celestial longitude of prime meridian state name, specified as a character vector.

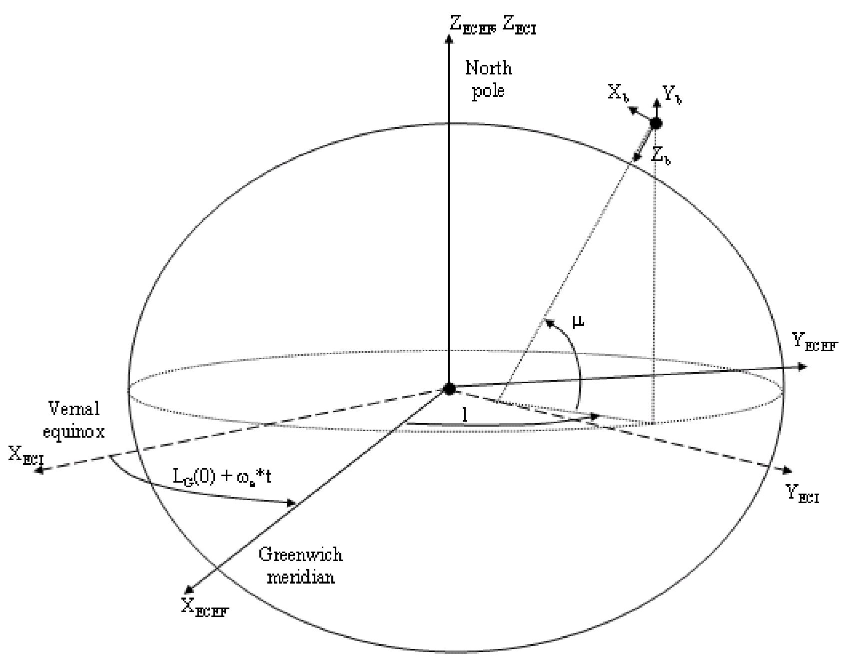
Dependencies

- To enable this parameter, set:**Planet model** to `Custom`.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich: e.g., 'LG'**.

Programmatic Use
Block Parameter: `LPM_statename`
Type: character vector
Values: `''` | scalar
Default: `''`

Algorithms

The origin of the ECEF coordinate frame is the center of the Earth. In addition, the body of interest is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The representation of the rotation of ECEF frame from ECI frame is simplified to consider only the constant rotation of the ellipsoid Earth (ω_e) including an initial celestial longitude ($L_G(0)$). This excellent approximation allows the forces due to the Earth's complex motion relative to the "fixed stars" to be neglected.



The translational motion of the ECEF coordinate frame is given below, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the body frame and the mass of the body m is assumed constant.

$$\bar{F}_b = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m\left(\dot{\bar{V}}_b + \bar{\omega}_b \times \bar{V}_b + DCM_{bf}\bar{\omega}_e \times \bar{V}_b + DCM_{bf}\left(\bar{\omega}_e \times \left(\bar{\omega}_e \times \bar{X}_f\right)\right)\right)$$

where the change of position in ECEF $\dot{x}_f$ is calculated by

$$\dot{x}_f = DCM_{fb}\bar{V}_b$$

and the velocity of the body with respect to ECEF frame, expressed in body frame ($\bar{V}_b$), angular rates of the body with respect to ECI frame, expressed in body frame ($\bar{\omega}_b$). Earth rotation rate ($\bar{\omega}_e$), and relative angular rates of the body with respect to north-east-down (NED) frame, expressed in body frame ($\bar{\omega}_{rel}$), are defined as

$$\bar{V}_b = \begin{bmatrix} u \\ v \\ w \end{bmatrix}, \bar{\omega}_{rel} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \bar{\omega}_e = \begin{bmatrix} 0 \\ 0 \\ \omega_e \end{bmatrix}, \bar{\omega}_b = \bar{\omega}_{rel} + DCM_{bf}\bar{\omega}_e + DCM_{be}\bar{\omega}_{ned}$$

$$\bar{\omega}_{ned} = \begin{bmatrix} \dot{l}\cos\mu \\ -\dot{\mu} \\ -\dot{l}\sin\mu \end{bmatrix} = \begin{bmatrix} V_E/(N+h) \\ -V_N/(M+h) \\ -V_E \bullet \tan\mu/(N+h) \end{bmatrix}$$

The rotational dynamics of the body defined in body-fixed frame are given below, where the applied moments are $[L\ M\ N]^T$, and the inertia tensor I is with respect to the origin O.

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{\omega}_b \end{bmatrix} = \frac{1}{m}\bar{F}_b - \left[\bar{\omega}_b \times \bar{V}_b + DCM_{bf}\bar{\omega}_e \times \bar{V}_b + DCM_{bf}\left(\bar{\omega}_e \times \left(\bar{\omega}_e \times \bar{X}_f\right)\right)\right]$$

$$A_{becef} = \frac{F_b}{m}$$

$$\bar{M}_b = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\bar{\omega}}_b + \bar{\omega}_b \times (I\bar{\omega}_b)$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The integration of the rate of change of the quaternion vector is given below.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -\,{}^1\!/_2 \begin{bmatrix} 0 & \omega_b(1) & \omega_b(2) & \omega_b(3) \\ -\omega_b(1) & 0 & -\omega_b(3) & \omega_b(2) \\ -\omega_b(2) & \omega_b(3) & 0 & -\omega_b(1) \\ -\omega_b(3) & -\omega_b(2) & \omega_b(1) & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Version History

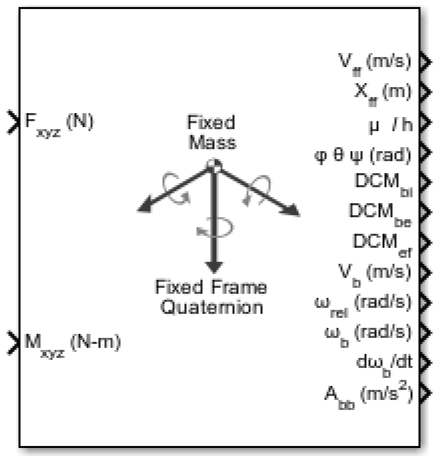
Introduced in R2006a

R2023b: 6DOF ECEF (Quaternion) Block Coordinate Frame Changes

Behavior changed in R2023b

These updates better clarify the coordinate frame of the 6DOF ECEF (Quaternion) block when **Central body** is `Custom`:

- New block icon.



- Port name subscripts of `ecef` have changed to `ff`.
- The **Celestial longitude of Greenwich source** programmatic name has changed from `LG0` to `LPM`. Existing scripts continue to work.
- Updated block parameter names, but models from previous releases continue to work.

Old Parameter Name or Setting	New Parameter Name
ECEF position...	**Fixed frame position**...
When **Planet model** is `Earth`	**Celestial longitude of Greenwich source** and **Celestial longitude of Greenwich [deg]**
When **Planet model** is `Custom`	**Celestial longitude of prime meridian source** and **Celestial longitude of prime meridian [deg]**

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation, 2nd ed*. Hoboken, NJ: John Wiley & Sons, 2003.

[2] McFarland, Richard E. "A Standard Kinematic Model for Flight simulation at NASA-Ames." NASA CR-2497.

[3] "Supplement to Department of Defense World Geodetic System 1984 Technical Report: Part I - Methods, Techniques and Data Used in WGS84 Development." DMA TR8350.2-A.

Extended Capabilities

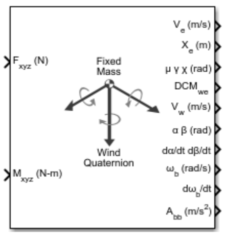
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

6DOF Wind (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion with respect to wind axes



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The 6DOF Wind (Quaternion) block considers the rotation of a wind-fixed coordinate frame (X_w, Y_w, Z_w) about an flat Earth reference frame (X_e, Y_e, Z_e). For more information on the wind-fixed coordinate frame, see “Algorithms” on page 5-124.

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Limitations

The block assumes that the applied forces act at the center of gravity of the body, and that the mass and inertia are constant.

Ports

Input

F_{xyz}(N) — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

μ γ χ (rad) — Wind rotation angles
three-element vector

Wind rotation angles [bank, flight path, heading], returned as a three-element vector, in radians.

Data Types: `double`

DCM_{we} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to wind-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_w — Velocity in wind-fixed frame
three-element vector

Velocity in wind-fixed frame, returned as a three-element vector.

Data Types: `double`

α β (rad) — Angle of attack and sideslip angle
two-element vector

Angle of attack and sideslip angle, returned as a two-element vector, in radians.

Data Types: `double`

dα/dt dβ/dt — Rate of change of angle of attack and rate of change of sideslip angle
two-element vector

Rate of change of angle of attack and rate of change of sideslip angle, returned as a two-element vector, in radians per second.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations in body-fixed axes
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this point, select **Include inertial acceleration**.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
Fixed	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
Simple Variable	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
Custom Variable	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Quaternion` (default) | `Wind Angles`

Equations of motion representation, specified according to the following table.

Representation	Description
Quaternion	Use quaternions within equations of motion.
Wind Angles	Use wind angles within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-124.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Wind Angles` | `Quaternion`
Default: `'Wind Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial airspeed, angle of attack, and sideslip angle [V,alpha,beta] — Initial airspeed, angle of attack, and sideslip angle

`[0 0 0]` (default) | three-element vector

Initial airspeed, angle of attack, and sideslip angle, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial wind orientation [bank angle,flight path angle,heading angle] — Initial wind orientation

`[0 0 0]` (default) | three-element vector

Initial wind angles [bank, flight path, and heading], specified as a three-element vector in radians.

Programmatic Use
Block Parameter: `wind_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Inertia in body axis — Inertia of body

`eye(3)` (default) | scalar

Inertia of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `inertia`
Type: character vector
Values: `'eye(3)'` | double scalar
Default: `'eye(3)'`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state names, specified as a character vector.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

`''` (default) | character vector

Incidence angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector
Values: `''`
Default: `''`

Sideslip angle e.g., 'beta' — Sideslip angle state name

`''` (default) | character vector

Sideslip angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `beta_statename`
Type: character vector
Values: `''`
Default: `''`

Wind orientation e.g., {'mu', 'gamma', 'chi'} — Wind orientation state names

`''` (default) | comma-separated list surrounded by braces

Wind orientation state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `wind_statename`
Type: character vector
Values: `''`
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Mass: e.g., 'mass' — Mass state name

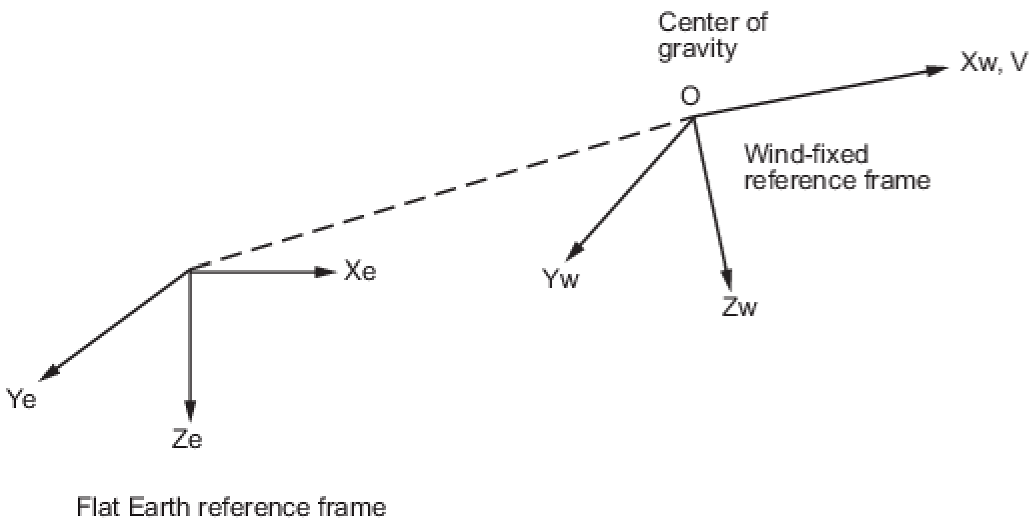
`''` (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The origin of the wind-fixed coordinate frame is the center of gravity of the body, and the body is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The flat Earth reference frame is considered inertial, an excellent approximation that allows the forces due to the Earth's motion relative to the “fixed stars” to be neglected.



The translational motion of the wind-fixed coordinate frame is given below, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the wind-fixed frame, and the mass of the body m is assumed constant.

$$\bar{F}_w = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m(\dot{\bar{V}}_w + \bar{\omega}_w \times \bar{V}_w)$$

$$A_{be} = DCM_{wb}\frac{\bar{F}_w}{m}$$

$$\bar{V}_w = \begin{bmatrix} V \\ 0 \\ 0 \end{bmatrix}, \bar{\omega}_w = \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = DCM_{wb}\begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}, \bar{\omega}_b = \begin{bmatrix} p_b \\ q_b \\ r_b \end{bmatrix}$$

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{w}_b \end{bmatrix} = DCM_{wb}\left[\frac{\bar{F}_w}{m} - \bar{\omega}_w \times \bar{V}_w\right]$$

The rotational dynamics of the body-fixed frame are given below, where the applied moments are $[L\ M\ N]^T$, and the inertia tensor I is with respect to the origin O. Inertia tensor I is easier to define in body-fixed frame.

$$\bar{M}_b = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\bar{\omega}}_b + \bar{\omega}_b \times (I\bar{\omega}_b)$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The integration of the rate of change of the quaternion vector is given below.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -\frac{1}{2}\begin{bmatrix} 0 & p & q & r \\ -p & 0 & -r & q \\ -q & r & 0 & -p \\ -r & -q & p & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. New York: John Wiley & Sons, 1992.

Extended Capabilities

C/C++ Code Generation

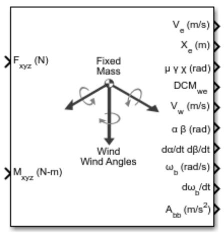
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

6DOF Wind (Wind Angles)

Implement wind angle representation of six-degrees-of-freedom equations of motion



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The 6DOF Wind (Wind Angles) block implements a wind angle representation of six-degrees-of-freedom equations of motion. For a description of the coordinate system employed and the translational dynamics, see the block description for the 6DOF Wind (Quaternion) block.

For more information on the relationship between the wind angles, see “Algorithms” on page 5-134

Limitations

The block assumes that the applied forces act at the center of gravity of the body, and that the mass and inertia are constant.

Ports

Input

F_{xyz}(N) — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

μ γ χ (rad) — Wind rotation angles
three-element vector

Wind rotation angles [bank, flight path, heading], returned as a three-element vector, in radians.

Data Types: `double`

DCM_{we} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to wind-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_w — Velocity in wind-fixed frame
three-element vector

Velocity in wind-fixed frame, returned as a three-element vector.

Data Types: `double`

α β (rad) — Angle of attack and sideslip angle
two-element vector

Angle of attack and sideslip angle, returned as a two-element vector, in radians.

Data Types: `double`

dα/dt dβ/dt — Rate of change of angle of attack and rate of change of sideslip angle
two-element vector

Rate of change of angle of attack and rate of change of sideslip angle, returned as a two-element vector, in radians per second.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations in body-fixed axes
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Wind Angles` (default) | `Quaternion`

Equations of motion representation, specified according to the following table.

Representation	Description
`Wind Angles`	Use wind angles within equations of motion.
`Quaternion`	Use quaternions within equations of motion.

The `Wind Angles` selection conforms to the equations of motion in “Algorithms” on page 5-134.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Wind Angles` | `Quaternion`
Default: `'Wind Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial airspeed, angle of attack, and sideslip angle [V,alpha,beta] — Initial airspeed, angle of attack, and sideslip angle

`[0 0 0]` (default) | three-element vector

Initial airspeed, angle of attack, and sideslip angle, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial wind orientation [bank angle,flight path angle,heading angle] — Initial wind orientation

`[0 0 0]` (default) | three-element vector

Initial wind angles [bank, flight path, and heading], specified as a three-element vector in radians.

Programmatic Use
Block Parameter: `wind_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Inertia in body axis — Inertia of body

`eye(3)` (default) | scalar

Inertia of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `inertia`
Type: character vector
Values: `'eye(3)'` | double scalar
Default: `'eye(3)'`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

'' (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., 'V' — Velocity state name

'' (default) | character vector

Velocity state names, specified as a character vector.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

'' (default) | character vector

Incidence angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector
Values: `''`
Default: `''`

Sideslip angle e.g., 'beta' — Sideslip angle state name

'' (default) | character vector

Sideslip angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `beta_statename`
Type: character vector
Values: `''`
Default: `''`

Wind orientation e.g., {'mu', 'gamma', 'chi'} — Wind orientation state names

'' (default) | comma-separated list surrounded by braces

Wind orientation state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `wind_statename`
Type: character vector
Values: `''`
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

'' (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

'' (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Mass: e.g., 'mass' — Mass state name

'' (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The relationship between the wind angles $[\mu\gamma\chi]^T$ can be determined by resolving the wind rates into the wind-fixed coordinate frame.

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} \dot{\mu} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\gamma} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} \cos\gamma & 0 & -\sin\gamma \\ 0 & 1 & 0 \\ \sin\gamma & 0 & \cos\gamma \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\chi} \end{bmatrix} \equiv J^{-1} \begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix}$$

Inverting J then gives the required relationship to determine the wind rate vector.

$$\begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix} = J \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} 1 & (\sin\mu\tan\gamma) & (\cos\mu\tan\gamma) \\ 0 & \cos\mu & -\sin\mu \\ 0 & \dfrac{\sin\mu}{\cos\gamma} & \dfrac{\cos\mu}{\cos\gamma} \end{bmatrix} \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix}$$

The body-fixed angular rates are related to the wind-fixed angular rate by the following equation.

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = DMC_{wb} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Using this relationship in the wind rate vector equations, gives the relationship between the wind rate vector and the body-fixed angular rates.

$$\begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix} = J \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} 1 & (\sin\mu\tan\gamma) & (\cos\mu\tan\gamma) \\ 0 & \cos\mu & -\sin\mu \\ 0 & \dfrac{\sin\mu}{\cos\gamma} & \dfrac{\cos\mu}{\cos\gamma} \end{bmatrix} DMC_{wb} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. New York: John Wiley & Sons, 1992.

Extended Capabilities

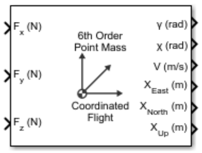
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

6th Order Point Mass (Coordinated Flight)

Calculate sixth-order point mass in coordinated flight



Libraries:
Aerospace Blockset / Equations of Motion / Point Mass

Description

The 6th Order Point Mass (Coordinated Flight) block performs the calculations for the translational motion of a single point mass or multiple point masses. For more information on the system for the translational motion of a single point mass or multiple mass, see “Algorithms” on page 5-139.

The 6th Order Point Mass (Coordinated Flight) block port labels change based on the input and output units selected from the **Units** list.

Limitations

- The block assumes that there is fully coordinated flight, i.e., there is no side force (wind axes) and sideslip is always zero.
- The flat Earth reference frame is considered inertial, an approximation that allows the forces due to the Earth motion relative to the "fixed stars" to be neglected.

Ports

Input

Port_1 — Force in *x*-axis
scalar | array

Force in *x*-axis, specified as a scalar or vector, in selected units.

Data Types: `double`

Port_2 — Force in *y*-axis
scalar | array

Force in *y*-axis, specified as a scalar or vector, in selected units.

Data Types: `double`

Port_3 — Force in *z*-axis
scalar | array

Force in *z*-axis, specified as a scalar or vector, in selected units.

Data Types: `double`

Output

Port_1 — Flight path angle
scalar | array

Flight path angle, returned as a scalar or vector, in radians.

Data Types: `double`

Port_2 — Heading angle
scalar | array

Heading angle, returned as a scalar or vector, in radians.

Data Types: `double`

Port_3 — Airspeed
scalar | array

Airspeed, returned as a scalar or vector, in selected units.

Data Types: `double`

Port_4 — Downrange or amount traveled east
scalar | array

Downrange or amount traveled east, returned as a scalar or vector, in selected units.

Data Types: `double`

Port_5 — Crossrange or amount travelled north
scalar | array

Crossrange or amount traveled north, returned as a scalar or vector, in selected units.

Data Types: `double`

Port_6 — Altitude or amount or travelled up
scalar | array

Altitude or amount traveled up, returned as a scalar or vector, in selected units.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Forces	Velocity	Position	Mass
`Metric (MKS)`	newtons	meters per second	meters	kilograms

Units	Forces	Velocity	Position	Mass
English (Velocity in ft/s)	pounds	feet per second	feet	slugs
English (Velocity in kts)	pounds	knots	feet	slugs

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Initial flight path angle — Initial flight path angle

`0` (default) | scalar | vector

Initial flight path angle of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `gamma0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial heading angle — Initial heading angle

`0` (default) | scalar | vector

Initial heading angle of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `chi0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial airspeed — Initial airspeed

`100` (default) | scalar | vector

Initial airspeed of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `V0`
Type: character vector
Values: scalar | vector
Default: `'100'`

Initial downrange [East] — Initial downrange

`0` (default) | scalar | vector

Initial downrange of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `x0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial crossrange [North] — Initial cross range

`0` (default) | scalar | vector

Initial crossrange of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `y0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial altitude [Up] — Initial altitude

`0` (default) | scalar | vector

Initial altitude of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `h0`
Type: character vector
Values: scalar | vector
Default: `'0'`

Initial mass — Point mass

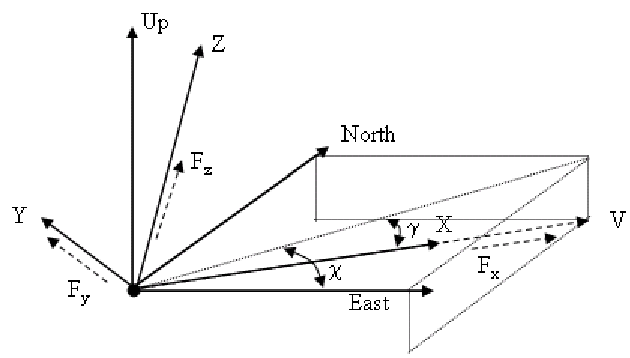
`1.0` (default) | scalar | vector

Mass of the point mass(es), specified as a scalar or vector.

Programmatic Use
Block Parameter: `mass0`
Type: character vector
Values: scalar | vector
Default: `'1.0'`

Algorithms

This figure shows the system for the translational motion of a single point mass or multiple point masses.



The translational motion of the point mass $[X_{East} X_{North} X_{Up}]^T$ are functions of airspeed (V), flight path angle (γ), and heading angle (χ),

$$F_x = m\dot{V}$$

$$F_y = (mV\cos\gamma)\dot{\chi}$$

$$F_z = mV\dot{\gamma}$$

$$\dot{X}_{East} = V\cos\chi\cos\gamma$$

$$\dot{X}_{North} = V\sin\chi\cos\gamma$$

$$\dot{X}_{Up} = V\sin\gamma$$

where the applied forces $[F_x F_y F_h]^T$ are in a system is defined by x-axis in the direction of vehicle velocity relative to air, z-axis is upward, and y-axis completes the right-handed frame, and the mass of the body m is assumed constant.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

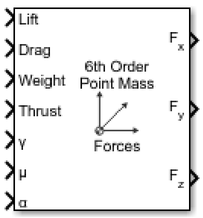
Generate C and C++ code using Simulink® Coder™.

See Also

4th Order Point Mass (Longitudinal) | 4th Order Point Mass Forces (Longitudinal) | 6th Order Point Mass Forces (Coordinated Flight) | 6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

6th Order Point Mass Forces (Coordinated Flight)

Calculate forces used by sixth-order point mass in coordinated flight



Libraries:
Aerospace Blockset / Equations of Motion / Point Mass

Description

The 6th Order Point Mass Forces (Coordinated Flight) block calculates the applied forces for a single point mass or multiple point masses. For more information on the system for the applied forces, see “Algorithms” on page 5-142.

Limitations

- The block assumes that there is fully coordinated flight, i.e., there is no side force (wind axes) and sideslip is always zero.
- The flat Earth reference frame is considered inertial, an approximation that allows the forces due to the Earth motion relative to the "fixed stars" to be neglected.

Ports

Input

Lift — Lift
scalar | array

Lift, specified as a scalar or array, in units of force.

Data Types: `double`

Drag — Drag
scalar | array

Drag, specified as a scalar or array, in units of force.

Data Types: `double`

Weight — Weight
scalar | array

Weight, specified as a scalar or array, in units of force.

Data Types: `double`

Thrust — Thrust
scalar | array

Thrust, specified as a scalar or array, in units of force.

Data Types: `double`

ɣ — Flight path angles
scalar | array

Flight path angle, specified as a scalar or array, in radians.

Data Types: `double`

μ — Bank angle
scalar | array

Bank angle, specified as a scalar or array, in radians.

Data Types: `double`

α — Angle of attack
scalar | array

Angle of attack, specified as a scalar or array, in radians.

Data Types: `double`

Output

$\mathbf{F_x}$ — Force in *x*- axis
scalar | array

Force in *x*-axis, specified as a scalar or array, in units of force.

Data Types: `double`

$\mathbf{F_y}$ — Force in *y*- axis
scalar | array

Force in *y*-axis, specified as a scalar or array, in units of force.

Data Types: `double`

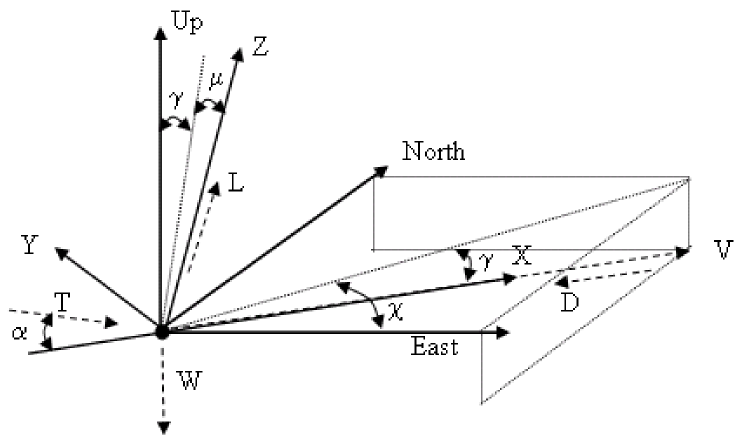
$\mathbf{F_z}$ — Force in *z*- axis
scalar | array

Force in *z*-axis, specified as a scalar or array, in units of force.

Data Types: `double`

Algorithms

This figure shows the applied forces in the system used by this block.



The applied forces $[F_x F_y F_h]^T$ are in a system is defined by x-axis in the direction of vehicle velocity relative to air, z-axis is upwards and y-axis completes the right-handed frame and are functions of lift (L), drag (D), thrust (T), weight (W), flight path angle (γ), angle of attack (α), and bank angle (μ).

$$F_x = T\cos\alpha - D - W\sin\gamma$$

$$F_y = (L + T\sin\alpha)\sin\mu$$

$$F_z = (L + T\sin\alpha)\cos\mu - W\cos\gamma$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

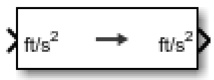
Generate C and C++ code using Simulink® Coder™.

See Also

4th Order Point Mass (Longitudinal) | 4th Order Point Mass Forces (Longitudinal) | 6th Order Point Mass (Coordinated Flight)

Acceleration Conversion

Convert from acceleration units to desired acceleration units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Acceleration Conversion block computes the conversion factor from specified input acceleration units to specified output acceleration units and applies the conversion factor to the input signal.

The Acceleration Conversion block port labels change based on the input and output units selected from the **Initial unit** and **Final unit** parameters.

Ports

Input

Port_1 — Acceleration
scalar | array

Acceleration, specified as a scalar or array, in initial acceleration units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Acceleration
scalar | array

Acceleration, returned as a scalar or array, in final acceleration units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`ft/s`^{s} (default) | `m/s`^{2} | `km/s`^{2} | `in/s`^{2} | `km/h-s` | `mph/s` | `G's`

Input units, specified as:

m/s^2	Meters per second squared
ft/s^2	Feet per second squared
km/s^2	Kilometers per second squared
in/s^2	Inches per second squared
km/h-s	Kilometers per hour per second
mph-s	Miles per hour per second
G's	g-units

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use

Block Parameter: `IU`
Type: character vector
Values: `'ft/s^s'` | `'m/s^2'` | `'km/s^2'` | `'in/s^2'` | `'km/h-s'` | `'mph/s'` | `'G's'`
Default: `'ft/s`^{2}`'`

Final unit — Output units

`ft/s^2'` (default) | `m/s^2` | `km/s^2` | `in/s^2` | `km/h-s` | `mph/s` | `G's`

Output units, specified as:

m/s^2	Meters per second squared
ft/s^2	Feet per second squared
km/s^2	Kilometers per second squared
in/s^2	Inches per second squared
km/h-s	Kilometers per hour per second
mph-s	Miles per hour per second
G's	g-units

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use

Block Parameter: `OU`
Type: character vector
Values: `'ft/s^2'` | `'m/s^2'` | `'km/s^2'` | `'in/s^2'` | `'km/h-s'` | `'mph/s'` | `'G's'`
Default: `'ft/s`^{2}`'`

Version History

Introduced before R2006a

Extended Capabilities

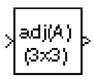
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

Adjoint of 3x3 Matrix

Compute adjoint of matrix



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Adjoint of 3x3 Matrix block computes the adjoint matrix for the input matrix. For related equations, see “Algorithms” on page 5-147.

Ports

Input

Port_1 — Input matrix
3-by-3 matrix

Input matrix, specified as a 3-by-3 matrix, in initial acceleration units.

Data Types: `double`

Output

Port_1 — Output acceleration
3-by-3 matrix

Output acceleration, returned as a 3-by-3 matrix, in final acceleration units.

Data Types: `double`

Algorithms

The input matrix has the form of

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

The adjoint of the matrix has the form of

$$adj(A) = \begin{pmatrix} +\begin{vmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{vmatrix} & -\begin{vmatrix} A_{12} & A_{13} \\ A_{32} & A_{33} \end{vmatrix} & +\begin{vmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \end{vmatrix} \\ -\begin{vmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{vmatrix} & +\begin{vmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{vmatrix} & -\begin{vmatrix} A_{11} & A_{13} \\ A_{21} & A_{23} \end{vmatrix} \\ +\begin{vmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{vmatrix} & -\begin{vmatrix} A_{11} & A_{12} \\ A_{31} & A_{32} \end{vmatrix} & +\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \end{pmatrix}$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

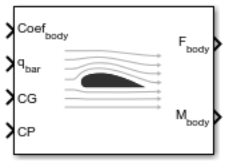
Generate C and C++ code using Simulink® Coder™.

See Also

Create 3x3 Matrix | Determinant of 3x3 Matrix | Invert 3x3 Matrix

Aerodynamic Forces and Moments

Compute aerodynamic forces and moments using aerodynamic coefficients, dynamic pressure, center of gravity, center of pressure, and velocity



Libraries:
Aerospace Blockset / Aerodynamics

Description

The Aerodynamic Forces and Moments block computes the aerodynamic forces and moments about the center of gravity.

The Aerodynamic Forces and Moments block port labels change based on the coordinate system selected from the **Input axes**, **Force axes**, and **Moment axes** list.

Limitations

- The default state of the block hides the V_b input port and assumes that the transformation is body-body.
- The center of gravity and the center of pressure are assumed to be in body axes.
- While this block has the ability to output forces and/or moments in the stability axes, the blocks in the Equations of Motion library are currently designed to accept forces and moments in either the body or wind axes only.

Ports

Input

Port_1 — Aerodynamic coefficients
six-element vector

Aerodynamic coefficients (in the chosen input axes) for forces and moments, specified as a vector. These coefficients are ordered into a vector depending on the choice of axes:

Input Axes	Input Vector
Body	(axial force C_x, side force C_y, normal force C_z, rolling moment C_l, pitching moment C_m, yawing moment C_n)
Stability	(drag force $C_{D(\beta=0)}$, side force C_y, lift force C_L, rolling moment C_l, pitching moment C_m, yawing moment C_n)
Wind	(drag force C_D, cross-wind force C_c, lift force C_L, rolling moment C_l, pitching moment C_m, yawing moment C_n)

Data Types: `double`

Port_2 — Dynamic pressure
scalar | three-element vector

Dynamic pressure, specified as a 1-by-3 array.

Data Types: `double`

Port_3 — Center of gravity
three-element vector

Center of gravity, specified as a 1-by-3 vector.

Data Types: `double`

Port_4 — Center of pressure
three-element vector

Center of pressure, specified as a 1-by-3 vector. This can also be taken as any general moment reference point as long as the rest of the model reflects the use of the moment reference point.

Data Types: `double`

Port_5 — Velocity in the body axes
three-element vector

Velocity in the body axes. specified as a 1-by-3 vector.

Dependencies

This port is enabled if the **Input axes** parameter is set to `Wind` or `Stability`.

Data Types: `double`

Output

Port_1 — Aerodynamic forces
three-element vector

Aerodynamic forces (in the chosen output axes), returned as three-element vector, at the center of gravity in *x*-, *y*-, and *z*-axes.

Data Types: `double`

Port_2 — Aerodynamic moments
three-element vector

Aerodynamic moments (in the chosen output axes), returned as three-element vector, at the center of gravity in *x*-, *y*-, and *z*-axes.

Data Types: `double`

Parameters

Input axes — Coordinate system for input coefficients

`Body` (default) | `Stability` | `Wind`

Coordinate system for input coefficients, specified as `Body` (default), `Stability`, or `Wind`.

Dependencies

Selecting `Stability` or `Wind` enables input port Port_5.

Programmatic Use
Block Parameter: `inputAxes`
Type: character vector
Values: `'Body'` | `'Stability'` | `'Wind'`
Default: `'Body'`

Force axes — Coordinate system for aerodynamic force

`Body` (default) | `Stability` | `Wind`

Coordinate system for aerodynamic force, specified as `Body` (default), `Stability`, or `Wind`.

Dependencies

Selecting `Stability` or `Wind` enables input port Port_5.

Programmatic Use
Block Parameter: `outputForcesAxes`
Type: character vector
Values: `'Body'` | `'Stability'` | `'Wind'`
Default: `'Body'`

Moment axes — Coordinate system for aerodynamic moment

`Body` (default) | `Stability` | `Wind`

Coordinate system for aerodynamic moment, specified as `Body` (default), `Stability`, or `Wind`.

Dependencies

Selecting `Stability` or `Wind` enables input port Port_5.

Programmatic Use
Block Parameter: `outputMomentAxes`
Type: character vector
Values: `'Body'` | `'Stability'` | `'Wind'`
Default: `'Body'`

Reference area — Reference area

`1` (default) | any double value

Reference area for calculating aerodynamic forces and moments, specified as any double value.

Programmatic Use
Block Parameter: `S`
Type: character vector
Values: any double value
Default: `'1'`

Reference span — Reference span

`1` (default) | any double value

Reference span for calculating aerodynamic moments in x-axes and z-axes, specified as any double value.

Programmatic Use
Block Parameter: `b`
Type: character vector
Values: any double value
Default: `'1'`

Reference length — Reference length

1 (default) | any double value

Reference length for calculating aerodynamic moment in the y-axes, specified as any double value.

Programmatic Use
Block Parameter: `cbar`
Type: character vector
Values: any double value
Default: `'1'`

Algorithms

Let α be the angle of attack and β the sideslip. The rotation from body to stability axes:

$$C_{s \leftarrow b} = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$

can be combined with the rotation from stability to wind axes:

$$C_{w \leftarrow s} = \begin{bmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

to yield the net rotation from body to wind axes:

$$C_{w \leftarrow b} = \begin{bmatrix} \cos(\alpha)\cos(\beta) & \sin(\beta) & \sin(\alpha)\cos(\beta) \\ -\cos(\alpha)\sin(\beta) & \cos(\beta) & -\sin(\alpha)\sin(\beta) \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$

Moment coefficients have the same notation in all systems. Force coefficients are given below. Note there are no specific symbols for stability-axes force components. However, the stability axes have two components that are unchanged from the other axes.

$$\mathbf{F}_A^w \equiv \begin{bmatrix} -D \\ -C \\ -L \end{bmatrix} = C_{w \leftarrow b} \cdot \begin{bmatrix} X_A \\ Y_A \\ Z_A \end{bmatrix} \equiv C_{w \leftarrow b} \cdot \mathbf{F}_A^b$$

Components/Axes	x	y	z
Wind	C_D	C_C	C_L

Components/Axes	***x***	***y***	***z***
Stability	—	C_Y	C_L
Body	C_X	C_Y	C_Z (–C_N)

Given these definitions, to account for the standard definitions of *D*, *C*, *Y* (where *Y* = -*C*), and *L*, force coefficients in the wind axes are multiplied by the negative identity *diag*(-1, -1, -1). Forces coefficients in the stability axes are multiplied by *diag*(-1, 1, -1). C_N and C_X are, respectively, the normal and axial force coefficients (C_N = -C_Z).

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation,* John Wiley & Sons, New York, 1992

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

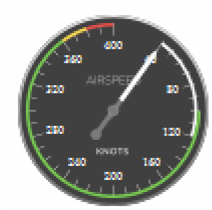
See Also

Dynamic Pressure | Digital DATCOM Forces and Moments | Estimate Center of Gravity | Moments about CG due to Forces

Topics
“NASA HL-20 Lifting Body Airframe” on page 3-14

Airspeed Indicator

Display measurements for aircraft airspeed



Libraries:
Aerospace Blockset / Flight Instruments

Description

The Airspeed Indicator block displays measurements for aircraft airspeed in knots.

By default, minor ticks represent 10-knot increments and major ticks represent 40-knot increments. The parameters **Minimum** and **Maximum** determine the minimum and maximum values on the gauge. The number and distribution of ticks is fixed, which means that the first and last tick display the minimum and maximum values. The ticks in between distribute evenly between the minimum and maximum values. For major ticks, the distribution of ticks is (**Maximum**-**Minimum**)/9. For minor ticks, the distribution of ticks is (**Maximum**-**Minimum**)/36.

The airspeed indicator has scale color bars that allow for overlapping for the first bar, displayed at a different radius. This different radius lets the block represent maximum speed with flap extended (V_{FE}) and stall speed with flap extended (V_{SO}) accurately for aircraft airspeed and stall speed.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name

Connect to signal for display, selected from list of signal names.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Minimum — Minimum tick mark value

40 (default) | finite | double | scalar

Minimum tick mark value, specified as a finite double scalar value, in knots.

Dependencies

The **Minimum** tick value must be less than the **Maximum** tick value.

Programmatic Use
Block Parameter: `Limits`
Type: double
Values: vector
Default: `[40 400]`, where `40` is the minimum value

Maximum — Maximum tick mark value

400 (default) | finite | double | scalar

Specify the maximum tick mark value, specified as a finite double scalar value, in knots.

Dependencies

The **Maximum** tick value must be greater than the **Minimum** tick value.

Programmatic Use
Block Parameter: `Limits`
Type: double
Values: vector
Default: `[40 400]`, where `400` is the maximum value

Scale Colors — Ranges of color bands
0 (default) | double | scalar

Ranges of color bands outside the scale, specified as a finite double scalar value. Specify the minimum and maximum color range to display on the gauge.

To add a new color, click `+`. To remove a color, click `-`.

Programmatic Use
Block Parameter: `ScaleColors`
Type: *n*-by-1 struct array
Values: struct array with elements `Min`, `Max`, and `Color`

Label — Name of connected signal

`Top` (default) | `Bottom` | `Hide`

Name of connected signal.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is ignored for code generation.

See Also

Altimeter | Artificial Horizon | Exhaust Gas Temperature (EGT) Indicator | Climb Rate Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Topics

“Display Measurements with Cockpit Instruments” on page 2-59
“Programmatically Interact with Gauge Band Colors” on page 2-61
“Flight Instrument Gauges” on page 2-58

Altimeter

Display measurements for aircraft altitude

Libraries:
Aerospace Blockset / Flight Instruments

Description

The Altimeter Indicator block displays the altitude above sea level in feet, also known as the pressure altitude. The block displays the altitude value with needles on a gauge and a numeric indicator.

- The gauge has 10 major ticks. Within each major tick are five minor ticks. This gauge has three needles. Using the needles, the altimeter can display accurately only altitudes between 0 and 100,000 feet.
  - For the longest needle, an increment of a small tick represents 20 feet and a major tick represents 100 feet.
  - For the second longest needle, a minor tick represents 200 feet and a major tick represents 1,000 feet.
  - For the shortest needle a minor tick represents 2,000 feet and a major tick represents 10,000 feet.
- For the numeric display, the block shows values as numeric characters between 0 and 9,999 feet. When the numeric display value reaches 10,000 feet, the gauge displays the value as the remaining values below 10,000 feet. For example, 12,345 feet displays as 2,345 feet. When a value is less than 0 (below sea level), the block displays `0`. The needles show the appropriate value except for when the value is below sea level or over 100,000 feet. Below sea level, the needles are set to `0`, over 100,000, the needles stay set at 100,000.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name

Connect to signal for display, selected from list of signal names.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is ignored for code generation.

See Also

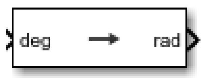
Airspeed Indicator | Artificial Horizon | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Topics

"Display Measurements with Cockpit Instruments" on page 2-59
"Flight Instrument Gauges" on page 2-58

Angle Conversion

Convert from angle units to desired angle units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Angle Conversion block computes the conversion factor from specified input angle units to specified output angle units and applies the conversion factor to the input signal.

The Angle Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Angle
scalar | array

Angle, specified as a scalar or array, in initial acceleration units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Angle
scalar | array

Angle, returned as a scalar, in final acceleration units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`deg` (default) | `rad` | `rev`

Input units, specified as:

deg	Degrees
rad	Radians
rev	Revolutions

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `'deg'` | `'rad'` | `'rev'`
Default: `'deg'`

Final unit — Output units

`rad` (default) | `deg` | `rev`

Output units, specified as:

deg	Degrees
rad	Radians
rev	Revolutions

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `'deg'` | `'rad'` | `'rev'`
Default: `'rad'`

Version History
Introduced before R2006a

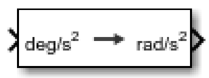
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Acceleration Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

Angular Acceleration Conversion

Convert from angular acceleration units to desired angular acceleration units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Angular Acceleration Conversion block computes the conversion factor from specified input angular acceleration units to specified output angular acceleration units and applies the conversion factor to the input signal.

The Angular Acceleration Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Angular input acceleration
scalar | array

Angle, specified as a scalar, in initial acceleration units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Angular output acceleration
scalar | array

Angle, returned as a scalar, in final acceleration units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`deg/s`^{2} (default) | `rad/s`^{2} | `rpm/s`

Specifies the input units, specified as:

deg/s^2	Degrees per second squared
rad/s^2	Radians per second squared
rpm/s	Revolutions per minute per second

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `'deg/s^2'` | `'rad/s^2'` | `'rpm/s'`
Default: `'deg/s^2'`

Final unit — Output units

rad/s^s (default) | deg/s^s | rpm/s

Output units, specified as:

deg/s^2	Degrees per second squared
rad/s^2	Radians per second squared
rpm/s	Revolutions per minute per second

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `'deg/s^2'` | `'rad/s^2'` | `'rpm/s'`
Default: `'rad/s^2'`

Version History
Introduced before R2006a

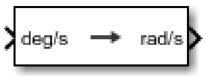
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Acceleration Conversion | Angle Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

Angular Velocity Conversion

Convert from angular velocity units to desired angular velocity units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Angular Velocity Conversion block computes the conversion factor from specified input angular velocity units to specified output angular velocity units and applies the conversion factor to the input signal.

The Angular Velocity Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Angular acceleration
scalar | array

Angular acceleration, specified as a scalar, in initial angular acceleration units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Angular acceleration
scalar | array

Angular acceleration, returned as a scalar, in final angular acceleration units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`deg/s` (default) | `rad/s` | `rpm`

Input units, specified as:

deg/s	Degrees per second
rad/s	Radians per second
rpm	Revolutions per minute

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `'deg/s'` | `'rad/s'` | `'rpm/s'`
Default: `'deg/s'`

Final unit — Output units

`rad/s` (default) | `deg/s` | `rpm`

Output units, specified as:

deg/s	Degrees per second
rad/s	Radians per second
rpm	Revolutions per minute

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `'deg/s'` | `'rad/s'` | `'rpm/s'`
Default: `'deg/s'`

Version History
Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

Artificial Horizon

Represent aircraft attitude relative to horizon

Libraries:
Aerospace Blockset / Flight Instruments

Description

The Artificial Horizon block represents aircraft attitude relative to horizon and displays roll and pitch in degrees:

- Values for roll cannot exceed +/– 90 degrees.
- Values for pitch cannot exceed +/– 30 degrees.

If the values exceed the maximum values, the gauge maximum and minimum values do not change.

Changes in roll value affect the gauge semicircles and the ticks located on the black arc turn accordingly. Changes in pitch value affect the scales and the distribution of the semicircles.

Combine the roll and pitch signals in a Mux block in the order:

1. Roll
2. Pitch

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name | 2-element signal

Connect to 2-element signal for display, selected from list of signal names. The 2-element signal consists of roll and pitch combined together in a Mux block, in degrees. You connect and display this combined signal. This input cannot be a bus signal.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

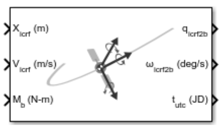
Airspeed Indicator | Altimeter | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Topics

“Display Measurements with Cockpit Instruments” on page 2-59
“Flight Instrument Gauges” on page 2-58

Attitude Dynamics

Model attitude dynamics of one or more spacecraft



Libraries:
Aerospace Blockset / Spacecraft / Spacecraft Dynamics

Description

The Attitude Dynamics block models rotational dynamics of spacecraft using numerical integration. The block computes the attitude and angular velocity of one or more spacecraft over time using provided position and velocity states. For the most accurate results, use a variable-step solver with low tolerance settings (less than 1e-8). To trade off accuracy for speed, use larger tolerances, depending on your mission requirements.

For more information on the coordinate systems and rotational and rotation dynamics the Attitude Dynamics block uses, see "Algorithms" on page 5-185.

Ports

Input

X — Position of spacecraft
3-element vector | *numSat*-by-3 array

Position of the spacecraft with respect to the state vector coordinate frame, specified as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

- To change the coordinate frame for this port, set the **State vector output coordinate frame** parameter.
- The size of the initial conditions provided in the **Mass** or **Attitude** tab control the port dimension.

Data Types: `double`

V — Velocity
3-element vector | *numSat*-by-3 array

Velocity of the spacecraft with respect to the state vector coordinate frame, specified as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

- To change the coordinate frame for this port, set the **State vector output coordinate frame** parameter.
- The size of the initial conditions provided in the **Mass** or **Attitude** tab control the port dimension.

Data Types: `double`

M_b — External moments
3-element vector | *numSat*-by-3 array

Moment applied to the spacecraft in the body frame, specified as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

To enable this port, select the **Input body moments** check box.

Data Types: `double`

dm/dt — Rate of change of mass
scalar | 1D array of size *numSat*

Rate of change of mass (positive if accreted, negative if ablated) at the current timestep, specified as a scalar or 1D array of size *numSat*. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Simple Variable`.

Data Types: `double`

m — Spacecraft mass
scalar | 1D array of size *numSat*

Spacecraft mass at the current timestep. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

I — Spacecraft inertia tensor
3-by-3 array | 3-by-3-by-*numSat* array

Spacecraft inertia tensor, specified as a 3-by-3 array or 3-by-3-by-*numSat* array at the current timestep. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 array | 3-by-3-by-*numSat* array

Rate of change of inertia tensor matrix, specified as a 3-by-3 array or 3-by-3-by-*numSat* array at the current time step. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

V_{re} — Relative velocity
3-element vector | *numSat*-by-3 array

Relative velocity at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a 3-element vector or *numSat*-by-3 array. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set **Mass type** to `Custom Variable` or `Simple Variable`.
- Select the **Include mass flow relative velocity** check box.

Data Types: `double`

φθψ — Moon libration angles
3-element vector

Moon libration angles for transformation between the International Celestial Reference Frame (ICRF) and Moon-centric fixed-frame using the Moon-centric Principal Axis (PA) system, specified as a 3-element vector. To get these values, use the Moon Libration block.

Note The fixed-frame used by this block when **Central body** is set to `Moon` is the Mean Earth/pole axis (ME) system. For more information, see “Algorithms” on page 5-669.

Dependencies

To enable this port:

- Set **Central body** to `Moon`.
- Select the **Input Moon libration angles** check box.

Data Types: `double`

αδW — Right ascension, declination, and rotation angle
3-element vector

Central body spin axis instantaneous right ascension, declination, and rotation angle, specified as a 3-element vector. This port is available only for custom central bodies.

Dependencies

To enable this port:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Port`.

Data Types: `double`

Output

$q_{body2icrf}$ — Spacecraft attitude quaternion
4-element quaternion | *numSat*-by-4 array

Spacecraft attitude quaternion, returned as a scalar-first 4-element quaternion rotation from the body axis to the outport frame or scalar-first *numSat*-by-4 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

The coordinate frame and attitude format of this port depends on these settings:

- To specify the attitude reference coordinate frame, set the **Attitude reference coordinate frame** parameter.
- Set **Attitude representation** to `Quaternion`.

Data Types: `double`

DCM — Spacecraft attitude direction cosine matrix
3-by-3 array | *numSat*-by-3-by-3 array

Spacecraft attitude direction cosine matrix (DCM), returned as a 3-by-3 array or *numSat*-by-3-by-3 array. *numSat* is the number of spacecraft.

Dependencies

The coordinate frame and attitude format of this port depends on these settings:

- To specify the attitude reference coordinate frame, set the **Attitude reference coordinate frame** parameter.
- Set **Attitude representation** to `DCM`.

Data Types: `double`

R1,R2,R3 — Spacecraft attitude Euler angles
3-element vector | *numSat*-by-3 array

Spacecraft attitude Euler angles, returned as a 3-element vector or *numSat*-by-3 array. *numSat* is the number of spacecraft.

Dependencies

The coordinate frame and attitude format of this port depend on these settings:

- To specify the attitude reference coordinate frame, set the **Attitude reference coordinate frame** parameter.
- Set **Attitude representation** to `Euler angles`.

Data Types: `double`

ω — Angular rate of spacecraft
3-element vector | *numSat*-by-3 array

Angular rate of the spacecraft relative to the attitude reference coordinate frame, returned as a 3-element vector or *numSat*-by-3 array, expressed as body axis angular rates PQR. *numSat* is the number of spacecraft.

Dependencies

The attitude reference coordinate frame depends on the **Attitude reference coordinate frame** parameter.

Data Types: `double`

$q_{icrf2ff}$ — Coordinate system transformation
4-element array

Coordinate system transformation between the ICRF and fixed-frame coordinate system at the current timestep, returned as a 4-element array.

Dependencies

To enable this port, select the **Output quaternion (ICRF to Fixed-frame)** check box.

Data Types: `double`

t_{utc} — Time at current time step
scalar | 6-element array

Time at current time step, returned as a:

- scalar — If you specify the **Start data/time** parameter as a Julian date.
- 6-element array — If you specify the **Start data/time** parameter as a Gregorian date with six elements: year, month, day, hours, minutes, seconds.

This value equals the **Start date/time** parameter value plus the elapsed simulation time.

Dependencies

To enable this parameter, select the **Output current date/time (UTC Julian date)** check box.

Data Types: `double`

Fuel Status — Fuel status
scalar | *numSat*-element array

Fuel tank status at the current timestep, returned as a scalar or *numSat*-element array:

- `1` — Tank is full.
- `0` — Tank is not full or empty.
- `-1` — Tank is empty.

numSat is the number of spacecraft.

Dependencies

To enable this port:,

- Set **Mass type** to `Simple Variable`.
- Select the **Output fuel tank status** check box.

Data Types: `double`

Parameters

Main

Input body moments — Option to enable external moments

on (default) | off

To enable external moments to be included in the integration of the spacecraft equations of motion in the body frame, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `momentsIn`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

State vector coordinate frame — Position and velocity state port coordinate frame

`ICRF` (default) | `Fixed-frame`

Position and velocity state port coordinate frame setup, specified as `ICRF` or `Fixed-frame`.

Programmatic Use
Block Parameter: `outportFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'`
Default: `'ICRF'`

Start date/time (UTC Julian date) — Initial start time for simulation

`juliandate (2020, 1, 1, 12, 0, 0)` (default) | valid scalar Julian date | valid Gregorian date

Initial start date and time of simulation, specified as a Julian or Gregorian date. A valid Gregorian date must include the year, month, day, hours, minutes, and seconds as a 1D or 6-element array. The block defines initial conditions using this value.

Tip To calculate the Julian date, use the `juliandate` function.

Tunable: Yes

Dependencies

The data format for this parameter is controlled by the **Time format** parameter.

Programmatic Use
Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1, 12, 0, 0)'` | valid scalar Julian date | valid Gregorian date
Default: `'juliandate(2020, 1, 1, 12, 0, 0)'`

Output current date/time (UTC Julian date) — Option to add output port t_{utc}

on (default) | off

To output the current date or time, select this check box. Otherwise, clear this check box.

Dependencies

The data format for this parameter is controlled by the **Time format** parameter.

Programmatic Use
Block Parameter: `dateOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Action for out-of-range input — Out-of-range block behavior

`Warning` (default) | `Error` | `None`

Out-of-range block behavior action. Specify one of these options.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Mass

Mass type — Spacecraft mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Spacecraft mass type, specified as:

- `Fixed` — Mass and inertia are constant throughout the simulation.
- `Simple Variable` — Mass and inertia vary linearly as a function of mass rate.
- `Custom Variable` — Instantaneous mass, inertia, and inertia rate are inputs to the block.

Programmatic Use
Block Parameter: `massType`
Type: character vector
Values: `'Fixed'` | `'Simple Variable'` | `'Custom Variable'`
Default: `'Fixed'`

Mass — Initial mass of rigid body spacecraft

4.0 (default) | scalar | vector of size *numSat*

Initial mass of rigid body spacecraft, specified as scalar or vector of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set the **Mass type** parameter to either **Fixed** or **Simple variable**.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'4.0'`

Empty mass — Spacecraft empty mass

`3.5` (default) | scalar | vector of size *numSat*

Spacecraft empty (dry) mass, specified as a scalar or as a vector of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `emptyMass`
Type: character vector
Values: 1D array of size *numSat* | 1D array of size *numSat*
Default: `'3.5'`

Full mass — Spacecraft full mass

4.0 (default) | scalar | vector of size *numSat*

Spacecraft full (wet) mass, specified as a scalar or as a vector of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `fullMass`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'4.0'`

Inertia tensor — Inertia tensor matrix

`[0.2273, 0, 0; 0 0.2273 0; 0 0 .0040]` (default) | 3-by-3 array | 3-by-3-by-*numSat* array

Initial inertia tensor matrix of the spacecraft, specified, as a 3-by-3 array for a single spacecraft or a 3-by-3-by-*numSat* array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use

Block Parameter: `inertia`
Type: character vector
Values: `'[0.2273, 0, 0; 0 0.2273 0; 0 0 .0040]'` | 3-by-3 array | 3-by-3-by-*numSat* array
Default: `'[0.2273, 0, 0; 0 0.2273 0; 0 0 .0040]'`

Empty inertia tensor — Empty inertia tensor matrix

`[0.1989, 0, 0; 0 0.1989 0; 0 0 .0035]` (default) | 3-by-3 array | 3-by-3-by-*numSat* array

Empty (dry) inertia tensor matrix, specified as a 3-by-3 array for a single spacecraft or a 3-by-3-by-*numSat* array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use

Block Parameter: `emptyInertia`
Type: character vector
Values: 3-by-3 array | 3-by-3-by-*numSat* array
Default: `[0.1989, 0, 0; 0 0.1989 0; 0 0 .0035]`

Full inertia tensor — Full inertia tensor matrix

`[0.2273, 0, 0; 0, 0.2273, 0; 0, 0, .0040]` (default) | 3-by-3 array | 3-by-3-by-*numSat* array

Full (wet) inertia tensor matrix, specified as a 3-by-3 array for a single spacecraft or a 3-by-3-by-*numSat* array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use

Block Parameter: `fullInertia`
Type: character vector
Values: 3-by-3 array | 3-by-3-by-*numSat* array
Default: `[0.2273, 0, 0; 0, 0.2273, 0; 0, 0, .0040]`

Include mass flow relative velocity — Option to enable mass flow velocity

off (default) | on

To enable mass flow velocity to the block, select this check box. The mass flow velocity is the relative velocity in the body frame at which the mass is accreted or ablated. To disable mass flow velocity to the block, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple variable` or `Custom variable`.

Programmatic Use
Block Parameter: `useMassFlowRelativeVelocity`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Limit mass flow when mass is empty or full — Option to limit mass flow

on (default) | off

To limit the mass flow when the spacecraft mass is full or empty, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `limitMassFlow`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Output fuel tank status — Option to enable fuel tank status

on (default) | off

To enable fuel tank status, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `outputFuelStatus`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Attitude

Attitude reference coordinate frame — Attitude and angular rate coordinate frame

`ICRF` (default) | `Fixed-frame` | `NED` | `LVLH`

Attitude and angular rate coordinate frame with respect to the attitude and angular rate initial conditions, specified as:

- `ICRF`
- `Fixed-frame`
- `NED`

- `LVLH`

Programmatic Use
Block Parameter: `attitudeFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'` | `'NED'` | `'LVLH'`
Default: `'ICRF'`

Attitude representation — Orientation format

`Quaternion` (default) | `DCM` | `Euler angles`

Orientation format for spacecraft attitude (initial condition and output port), specified as `Quaternion`, `DCM`, or `Euler angles`.

Programmatic Use
Block Parameter: `attitudeFrame`
Type: character vector
Values: `'Quaternion'` | `'DCM'` | `'Euler angles'`
Default: `'Quaternion'`

Initial body attitude — Spacecraft initial attitude

`[1, 0, 0, 0]` (default) | 4-element vector | *numSat*-by-4 array | 3-by-3 array | *numSat*-by-3-by-3 array | 3-element vector | *numSat*-by-3 array

Spacecraft initial attitude (orientation) of the spacecraft provided as either a quaternion, DCM, or Euler angle set with respect to **Attitude representation**.

Tunable: Yes

Dependencies

This parameter name and value format changes depending on the **Attitude representation** parameter.

Parameter Name	Attitude Representation Setting	Value Format
Initial quaternion	`Quaternion`	• 4-element vector • *numSat*-by-4 array
Initial DCM	`DCM`	• 3-by-3 array • *numSat*-by-3-by-3 array
Initial Euler angles	`Euler angles`	• 3-element vector • *numSat*-by-3 array

Programmatic Use
Block Parameter: `attitude`
Type: character vector
Values: 4-element vector | *numSat*-by-4 array | 3-by-3 array | *numSat*-by-3-by-3 array | 3-element array | *numSat*-by-3 array
Default: `'[1, 0, 0, 0]'`

Angle rotation order — Angle rotation order

ZYX (default) | ZYX | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation angle sequence for Euler angle attitude representation.

Tunable: Yes

Dependencies

To enable this parameter, set **Attitude representation** to `Euler angles`.

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: `'ZYX'` | `'ZYZ'` |`'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`
Default: `'ZYX'`

Initial body angular rates PQR — Initial body-fixed angular rates

`[0, 0, 0]` (default) | 3-element vector | *numSat*-by-3 array

Initial body-fixed angular rates (PQR) with respect to **Attitude reference coordinate frame**.

Tunable: Yes

Programmatic Use
Block Parameter: `attitudeRate`
Type: character vector
Values: | 3-element vector | *numSat*-by-3 array
Default: `[0, 0, 0]`

Include gravity gradient torque — Option to enable gravity gradient torque

on (default) | off

Select this check box to enable the use of the gravity gradient torque in the block rotational dynamics equations. Otherwise, clear this check box.

Tunable: Yes

Programmatic Use
Block Parameter: `angAccelOut`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Central Body

Central body — Celestial body around which spacecraft orbits

`Earth` (default) | `Moon` | `Mercury` | `Venus` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Custom`

Celestial body around which the spacecraft orbits, specified as `Earth`, `Moon`, `Mercury`, `Venus`, `Mars`, `Jupiter`, `Saturn`, `Uranus`, `Neptune`, or `Custom`.

Programmatic Use
Block Parameter: `centralBody`
Type: character vector
Values: `'Earth'` | `'Moon'` |`'Mercury'` | `'Venus'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Custom'` |
Default: `'Earth'`

Rotational rate — Rotational rate

`4.06124975e-3` (default) | scalar

Rotational rate of a custom central body, specified as a scalar.

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `'customOmega'`
Type: character vector
Values: `'4.06124975e-3'` | scalar
Default: `'4.06124975e-3'`

Use Earth orientation parameters (EOPs) — Option to use Earth orientation parameters

on (default) | off

Select this check box to use Earth orientation parameters for the transformation between the ICRF and fixed-frame coordinate systems. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Central body** to `Earth`.

Additionally, it must satisfy one of these criteria:

- **State vector output coordinate frame** is set to `Fixed-frame`.
- **Attitude reference coordinate frame** is set to `Fixed-frame` or `NED`.

Programmatic Use
Block Parameter: `useEOPs`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

IERS EOP data file — Earth orientation data

`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

Dependencies

To enable this parameter:

- Select the **Use Earth orientation parameters (EOPs)** check box.

- Set **Central body** to `Earth`.

Programmatic Use
Block Parameter: `eopFile`
Type: character vector
Values: `'aeroiersdata.mat'` | MAT-file
Default: `'aeroiersdata.mat'`

Input Moon libration angles — Moon libration Euler angle rate

off (default) | on

To specify Euler libration angles (φ θ ψ) for Moon orientation, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Central body** to `Moon`.

Programmatic Use
Block Parameter: `useMoonLib`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Output quaternion (ICRF to Fixed-frame) — Option to add output transformation quaternion port

off (default) | on

To add an output transformation quaternion port for the quaternion transformation from the ICRF to the fixed-frame coordinate system, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `outputTransform`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Central body spin axis source — Central body spin source

`Port` (default) | `Dialog`

Central body spin axis source, specified as `Port` or `Dialog`. The block uses the spin axis to calculate the transformation from the ICRF to the fixed-frame coordinate system for the custom central body.

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `cbPoleSrc`
Type: character vector
Values: `'Port'` | `'Dialog'`
Default: `'Port'`

Spin axis right ascension (RA) at J2000 — Right ascension of central body spin axis at J2000

`317.68143` (default) | double scalar

Right ascension of central body spin axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRA`
Type: character vector
Values: `'317.68143'` | double scalar
Default: `'317.68143'`

Spin axis RA rate (deg/century) — Right ascension rate of central body spin axis

`-0.1061` (default) | double scalar

Right ascension rate of the central body spin axis, specified as a double scalar, in specified angle units per century.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRARate`
Type: character vector
Values: `'-0.1061'` | double scalar
Default: `'-0.1061'`

Spin axis declination (Dec) at J2000 — Declination of central body spin axis at J2000

`52.88650` (default) | double scalar

Declination of the central body spin axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.

- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbDec`
Type: character vector
Values: `'52.88650'` | double scalar
Default: `'52.88650'`

Spin axis Dec rate (deg/century) — Declination rate of central body spin axis

`-0.0609` (default) | double scalar

Declination rate of the central body spin axis, specified as a double scalar, in specified angle units per century.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbDecRate`
Type: character vector
Values: `'-0.0609'` | double scalar
Default: `'-0.0609'`

Initial rotation angle at J2000 — Rotation angle of central body *x*-axis

`176.630` (default) | double scalar

Rotation angle of the central body *x*-axis with respect to the ICRF *x*-axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar, in specified angle units.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRotAngle`
Type: character vector
Values: `'176.630'` | double scalar
Default: `'176.630'`

Rotation rate (deg/day) — Rotation rate of central body *x*-axis

`350.89198226` (default) | double scalar

Rotation rate of the central body *x*-axis with respect to the ICRF *x*-axis (2451545.0 JD, 2000 Jan 1 12:00:00 UTC), specified as a double scalar, in angle units per day.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRotRate`
Type: character vector
Values: `'350.89198226'` | double scalar
Default: `'350.89198226'`

Equatorial radius — Equatorial radius

`3396200` (default) | double scalar

Equatorial radius for a custom central body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Central Body** to `Custom`.

Programmatic Use
Block Parameter: `customR`
Type: character vector
Values: `'3396200'` | double scalar
Default: `'3396200'`

Flattening — Flattening ratio

`0.00589` (default) | double scalar

Flattening ratio for custom central body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `customF`
Type: character vector
Values: `'0.00589'` | double scalar
Default: `'0.00589'`

Gravitational parameter — Gravitational parameter

`4.305e13` (default) | double scalar

Gravitational parameter for a custom central body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `customMu`
Type: character vector
Values: `'4.305e13'` | double scalar
Default: `'4.305e13'`

Units

Units — Parameter and port units

`Metric (m/s)` (default) | `Metric (km/s)` | `Metric (km/h)` | `English (ft/s)` | `English (kts)`

Parameter and port units, specified as shown in the table.

Units	Moment	Mass	Inertia	Distance	Velocity	Acceleration
`Metric (m/s)`	Newton-meter	Kilograms	Kilogram m^2	meters	meters/sec	meters/sec^2
`Metric (km/s)`	Newton-meter	Kilograms	Kilogram m^2	kilometers	kilometers/sec	kilometers/sec^2
`Metric (km/h)`	Newton-meter	Kilograms	Kilogram m^2	kilometers	kilometers/hour	kilometers/$hour^2$
`English (ft/s)`	Foot-pound	Slugs	Slug ft^2	feet	feet/sec	feet/sec^2
`English (kts)`	Foot-pound	Slugs	Slug ft^2	nautical mile	knots	knots/sec

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (m/s)'` | `'Metric (km/s)'` | `'Metric (km/h)'` | `'English (ft/s)'` | `'English (kts)'`
Default: `'Metric (m/s)'`

Angle units — Angle units

`Degrees` (default) | `Radians`

Parameter and port units for angles, specified as `Degrees` or `Radians`.

Programmatic Use
Block Parameter: `angleUnits`
Type: character vector
Values: `'Degrees'` | `'Radians'`
Default: `'Degrees'`

Time format — Time format for start date and time output

`Julian date` (default) | `Gregorian`

Time format for **Start date/time (UTC Julian date)** and output port $\mathbf{t_{utc}}$, specified as `Julian date` or `Gregorian`.

Programmatic Use
Block Parameter: `timeFormat`
Type: character vector
Values: `'Julian date'` | `'Gregorian'`
Default: `'Julian date'`

Algorithms

Earth-Centric Coordinate Systems

The Attitude Dynamics block works in the ICRF and fixed-frame coordinate systems.

- ICRF — International Celestial Reference Frame. This frame can be treated as equal to the ECI coordinate system realized at J2000 (Jan 1 2000 12:00:00 TT). For more information, see “ECI Coordinates” on page 2-12.
- Fixed-frame — Fixed-frame is a generic term for the coordinate system that is fixed to the central body. The axes of the system rotate with the central body and are not fixed in inertial space. If the **Use Earth orientation parameters (EOPs)** check box is not selected, the block still uses the IAU2000/2005 reduction, but with Earth orientation parameters set to 0.

  - When **Central Body** is `Earth` and the **Use Earth orientation parameters (EOPs)** check box is selected, the fixed-frame coordinate system for the Moon is the Mean Earth/pole axis frame (ME). This frame is realized by two transformations. First, the values in the ICRF frame are transformed into the Principal Axis system (PA), which is the axis defined by the libration angles provided as inputs to the block (for more information, see Moon Libration). The states are then transformed into the ME system using a fixed rotation from the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [7].
  - When **Central Body** is `Moon` and the **Input Moon libration angles** check box is selected, the fixed-frame coordinate system for the Moon is the coordinate system defined by the libration angles provided as inputs to the block (for more information, see Moon Libration).
  - When **Central Body** is `Custom`, the fixed-frame coordinate system is defined by the poles of rotation and prime meridian defined by the block input α, δ, W, or the spin axis properties. In all other cases, the fixed-frame for each central body is defined by the directions of the poles of rotation and prime meridians defined in the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [7].

Vehicle-Centric Coordinate Systems

The Attitude Dynamics block system works in the Body frame, north-east-down (NED), and local vertical, local horizontal (LVLH) coordinate systems.

- Body frame — Fixed in both origin and orientation to the moving craft. For more information, see “Body Coordinates” on page 2-9.
- North-east-down — Noninertial system with its origin fixed at the aircraft or spacecraft center of gravity. For more information, see “NED Coordinates” on page 2-11.

- Local vertical, local horizontal — Also known as the spacecraft coordinate system, Gaussian coordinate system, or the orbit frame. LVLH is a rotating, accelerating frame commonly used in studies of relative motion, such as vehicle maneuvering. The axes of this frame are:
  - *R*-axis — Points outward from the spacecraft origin along its position vectors (with respect to the center of Earth). Measurements along this axis are referred to as radial.
  - *S*-axis — Completes the right hand coordinate system. This axis points in the direction of the velocity vector, but is only parallel to it for circular orbits. Measurements along this axis are referred to as along-track or transverse.
  - *W*-axis — Points normal to the orbital plane. Measurements along this axis are referred to as cross-track.

Rotational Dynamics

Rotational dynamics are governed by:

$$\dot{\vec{\omega}}_{b_{icrf}} = \left[\overrightarrow{M_b} - \vec{\omega}_{b_{icrf}} \times \left(I_{mom}\vec{\omega}_{b_{icrf}}\right) - \dot{I}_{mom}\vec{\omega}_{b_{icrf}}\right]\mathrm{inv}(I_{mom})$$

$$\dot{\vec{\omega}}_{b_{icrf}} \xRightarrow[\text{integrate}]{} \vec{q}_{b_{icrf}}, \vec{\omega}_{b_{icrf}},$$

where:

- $\vec{M}_b$ are the body moment components.
- I_{mom} is the spacecraft inertia tensor matrix.

When **Mass type** is `Fixed`, $\dot{I}_{mom}$ equals 0.

When **Mass type** is `Simple Variable`, this equation estimates the rate of change of the inertia tensor:

$$\dot{I}_{mom} = \frac{I_{full} - I_{empty}}{m_{full} - m_{empty}}\dot{m}$$

This equation gives the rate of change of the quaternion vector:

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} 0 & \omega_b(1) & \omega_b(2) & \omega_b(3) \\ -\omega_b(1) & 0 & -\omega_b(3) & \omega_b(2) \\ -\omega_b(2) & \omega_b(3) & 0 & -\omega_b(1) \\ -\omega_b(3) & -\omega_b(2) & \omega_b(1) & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Version History

Introduced in R2023a

References

[1] Vallado, David. *Fundamentals of Astrodynamics and Applications*. 4th ed. Hawthorne, CA: Microcosm Press, 2013.

[2] Vepa, Ranjan. *Dynamics and Control of Autonomous Space Vehicles and Robotics*. New York: Cambridge University Press, 2019.

[3] Stevens, Frank L., and Brian L. Stevens. *Aircraft Control and Simulation*. 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[4] Gottlieb, R. G. *Fast Gravity, Gravity Partials, Normalized Gravity, Gravity Gradient Torque and Magnetic Field: Derivation, Code and Data*. NASA Contractor Report 188243. Houston: NASA, February 1993.

[5] Seidelmann, P. Kenneth et al. "Report of the IAU/IAG Working Group on Cartographic Coordinates and Rotational Elements: 2006." *Celestial Mechanics and Dynamical Astronomy* 98 (2007): 155.

[6] Standish, E. M. "JPL Planetary and Lunar Ephemerides." DE405/LE405. Interoffice memorandum. JPL IOM 312.F-98-048. August 26, 1998.

Extended Capabilities

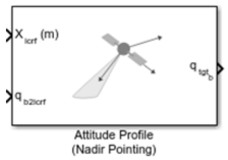
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Attitude Profile | Spacecraft Dynamics

Attitude Profile

Calculate shortest quaternion rotation



Libraries:
Aerospace Blockset / Spacecraft / Spacecraft Dynamics

Description

The Attitude Profile block calculates the shortest quaternion rotation that aligns the primary alignment vector with the primary constraint vector. A quaternion is defined using the scalar-first convention. Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Provide the primary constraint as either a pointing mode:

- `Point at nadir`
- `Point at celestial body`
- `Point at LatLonAlt`

or via a custom constraint vector. The block then aligns secondary alignment and constraint vectors as much as possible without breaking primary alignment.

The library contains three versions of the Attitude Profile block preconfigured for these common attitude control modes:

- Nadir Pointing — `Point at nadir`
- Geographic Pointing — `Point at LatLonAlt`
- Sun Tracking — `Point at celestial body` with Sun as the celestial target

For more information on the coordinate systems the Attitude Profile block uses, see “Algorithms” on page 5-194.

Ports

Input

X — Velocity state vector position
3-element vector

Position state vector of spacecraft at time t_{utc}.

Data Types: `double`

V — Velocity state vector
3-element vector

Velocity state vector of spacecraft at time t_{utc}, specified as a 3-element vector.

Dependencies

To enable this port, set **Constraint coordinate frame (CCF)** to `LVLH`.

Data Types: `double`

q — Spacecraft attitude
4-element vector

Attitude of the spacecraft at t_{utc}, represented as a quaternion from body frame to port coordinate frame, specified as a 4-element vector.

Data Types: `double`

t_{utc} — Current data or time
scalar

Current date or time, specified as a scalar, as a Julian date.

Dependencies

To enable this port, perform one of these:

- Set **Pointing mode** to `Point at celestial body` or `Point at LatLonAlt`
- Select the **Allow pointing mode change during run** check box.

Data Types: `double`

μ l — Geodetic latitude and longitude
2-element vector

Geodetic latitude and longitude (deg) of a terrestrial point of interest, specified as a 1-D array of size 2. This port is used together with altitude when **Pointing mode** is `Point at LatLongAlt`. This location is used as the primary constraint.

Dependencies

To enable this port, do one of these:

- Set **Pointing mode** to `LatLonAlt`.
- Select the **Allow pointing mode change during run** check box.

Data Types: `double`

h — Altitude
scalar

Altitude of terrestrial point of interest, specified as a scalar. This port is used together with geodetic latitude and longitude when **Pointing mode** is `Point at LatLongAlt`. This location is used as the primary constraint.

Dependencies

To enable this port, do one of these:

- Set **Pointing mode** to `LatLonAlt`.
- Select the **Allow pointing mode change during run** check box.

Data Types: `double`

$A1_b$ — Primary alignment vector
3-element vector

Primary alignment vector (in body frame), specified as a 3-element vector.

Dependencies

To enable this port, set **Primary alignment (body-frame)** to `Port`.

Data Types: `double`

$A2_b$ — Secondary alignment vector
3-element vector

Secondary alignment vector (in body frame), specified as a 3-element vector.

Dependencies

To enable this port, set **Secondary alignment (Body-frame)** to `Port`.

Data Types: `double`

$C1_{lvlh}$ — Primary constraint vector
3-element vector

Primary constraint vector, specified as a 3-element vector, in constraint coordinate frame.

Dependencies

To enable this port, set:

- **Pointing mode** to `Custom`.
- **Primary constraint (CCF)** to `Port`.

Data Types: `double`

$C2_{lvlh}$ — Secondary constraint
3-element vector

Secondary constraint vector, specified as a 3-element vector.

Dependencies

To enable this port, set **Secondary constraint (CCF)** to `Port`.

Data Types: `double`

Output

q_{tgt_b} — Quaternion rotation
4-element vector (scalar first)

Quaternion rotation by which to rotate from the spacecraft current orientation to the desired orientation (in body frame), returned as a 4-element vector.

Dependency

To enable this port, select the **Output rotation from current to updated attitude** check box.

Data Types: `double`

q_{new} — Updated spacecraft attitude
4-element vector

Updated spacecraft attitude with respect to **Port Coordinate Frame** following the quaternion rotation, returned as a 4-element vector.

Dependency

To enable this port, select the **Output attitude** check box.

Data Types: `double`

Parameters

Output rotation from current to updated attitude — Output rotation

on (default) | off

To output the rotation, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `outputError`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Output attitude — Output updated attitude

off (default) | on

To output the updated spacecraft attitude as a quaternion rotation from the body frame to the **Port coordinate frame**, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `outputFinalAttitude`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Port coordinate frame — Coordinate frame for position, velocity, and attitude ports

`ICRF` (default) | `Fixed-frame`

Coordinate frame for position, velocity, and attitude (**q**) ports. For more information about coordinate systems, see "Algorithms" on page 5-194.

Programmatic Use
Block Parameter: `portFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'`
Default: `'ICRF'`

Pointing mode — Primary vector alignment pointing mode

`Point at nadir` (default) | `Point at celestial body` | `Point at LatLonAlt` | `Custom`

Primary vector alignment pointing mode, specified as `Point at nadir`, `Point at celestial body`, `Point at LatLonAlt`, or `Custom`.

Programmatic Use
Block Parameter: `pointingMode`
Type: character vector
Values: `'Point at nadir'` | `'Point at celestial body'` | `'Point at LatLonAlt'` | `'Custom'`
Default: `'Point at nadir'`

Allow pointing mode change during run — Allow pointing mode change during run

off (default) | on

To allow pointing mode change during run, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `tunablePointing`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Celestial target — Celestial body

`Sun` (default) | `Mercury` | `Venus` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Pluto` | `Solar system barycenter` | `Earth-Moon barycenter`

Celestial body with which to align primary alignment vector.

Dependencies

To enable this parameter, set **Pointing mode** to `Point at celestial body`.

Programmatic Use
Block Parameter: `celestialTarget`
Type: character vector
Values: `'Sun'` | `'Mercury'` | `'Venus'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Pluto'` | `'Solar'` | `'Solar system barycenter'` | `'Earth-Moon barycenter'`
Default: `'Sun'`

Primary alignment (body-frame) — Primary alignment vector
`Dialog` (default) | `Port`

Primary alignment vector source, specified as `Port` or `Dialog`.

- `Port` — Specify port alignment array through the $\mathbf{A1_b}$ port.
- `Dialog` — Specify port alignment 3-element vector in the accompanying text box (default value of `[0 0 1]`).

Dependencies

To specify the port alignment array in a text box, set this parameter to `Dialog`.

Programmatic Use

Block Parameter: primaryAlignmentSrc | when primaryAlignmentSrc is 'Dialog', use primaryAlignment to set the primary alignment vector
Type: character vector
Values: 'Port' | 'Dialog' | primary alignment vector, specified 3-element vector
Default: 'Dialog'

Secondary alignment (body-frame) — Secondary alignment vector

Dialog (default) | Port

Secondary alignment vector source, specified as Port or Dialog.

- Port — Specify port alignment array through the **$A2_b$** port.
- Dialog — Specify port alignment 3-element vector in the accompanying text box (default value of [1 0 0]).

Dependencies

To specify the port alignment array in a text box, set this parameter to Dialog.

Programmatic Use

Block Parameter: secondaryAlignmentSrc | when secondaryAlignmentSrc is 'Dialog', use secondaryAlignment to set the secondary alignment vector
Type: character vector
Values: 'Port' | 'Dialog' | secondary alignment vector, specified as a 3-element vector
Default: 'Dialog'

Constraint coordinate frame, CCF — Constraint coordinate frame

ICRF (default) | Fixed-frame | LVLH | NED | Body-fixed

Coordinate frame in which constraint vectors are provided, specified as ICRF, Fixed-frame, LVLH, NED, or Body-fixed. For more information about coordinate systems, see “Algorithms” on page 5-194.

Programmatic Use

Block Parameter: constraintFrame
Type: character vector
Values: 'ICRF' | 'Fixed-frame' | 'LVLH' | 'NED' | 'Body-fixed'
Default: 'ICRF'

Primary constraint (CCF) — Primary constraint

Dialog (default) | Port

Primary constraint vector source, specified as Port or Dialog.

- Port — Specify primary constraint array through the **$C1_b$** port.
- Dialog — Specify port constraint 3-element vector in the accompanying text box (default value of [1 0 0]).

Dependencies

- To specify the port alignment array in a text box, set this parameter to Dialog.
- This parameter is affected when **Constraint coordinate frame (CCF)** is set to Custom.

Programmatic Use
Block Parameter: primaryConstraintSrc | when primaryConstraintSrc is 'Dialog', use primaryConstraint to set the primary constraint vector
Type: character vector
Values: 'Port' | 'Dialog' | primary constraint vector, specified as a 3-element vector
Default: 'Dialog'

Secondary constraint (CCF) — Secondary constraint
Dialog (default) | Port

Secondary constraint vector source, specified as Port or Dialog.

- Port — Specify secondary constraint array through the $\mathbf{C1_b}$ port.
- Dialog — Specify port constraint 3-element vector in the accompanying text box (default value of [0 1 0]).

After the primary alignment vector is aligned with the primary constraint vector, to fully define the rotation, the block attempts to align the secondary alignment vector with the rotation vector. The rotation vector should be the secondary constraint vector.

Whereas the primary constraint is enabled only for the custom pointing mode, the secondary constraint is always enabled.

Dependencies

To specify the port alignment array in a text box, set this parameter to Dialog.

Programmatic Use
Block Parameter: secondaryConstraintSrc | when secondaryConstraintSrc is 'Dialog', use secondaryConstraint to set the secondary constraint vector
Type: character vector
Values: 'Port' | 'Dialog' | secondary constraint vector, specified as a 3-element vector
Default: 'Dialog'

Algorithms

The Attitude Profile block uses Earth-centric and vehicle-centric coordinate systems.

Earth-Centric Coordinate Systems

The Earth-centric coordinate system uses the ICRF and fixed-frame coordinate systems:

- International Celestial Reference Frame. This frame can be treated as equal to the ECI coordinate system realized at J2000 (Jan 1 2000 12:00:00 TT. For more information, see “ECI Coordinates” on page 2-12.
- Fixed-frame — The fixed-frame for Earth this block uses is the International Terrestrial Reference Frame (ITRF). This reference frame is realized by the IAU2000/2006 reduction from the ICRF coordinate system. This frame is often described as the Earth-centered Earth-fixed reference frame.

Vehicle-Centric Coordinate Systems

The vehicle-centric coordinate system works in the Body frame, north-east-down (NED), and local vertical, local horizontal (LVLH) coordinate systems.

- Body frame — Fixed in both origin and orientation to the moving craft. For more information, see “Body Coordinates” on page 2-9.
- North-east-down — Noninertial system with its origin fixed at the aircraft or spacecraft center of gravity. For more information, see “NED Coordinates” on page 2-11.
- Local vertical, local horizontal — Also known as the spacecraft coordinate system, Gaussian coordinate system, or the orbit frame. LVLH is a rotating, accelerating frame commonly used in studies of relative motion, such as vehicle maneuvering. The axes of this frame are:

  - *R*-axis — Points outward from the spacecraft origin along its position vectors (with respect to the center of Earth). Measurements along this axis are referred to as radial.
  - *S*-axis — Completes the right hand coordinate system. This axis points in the direction of the velocity vector, but is only parallel to it for circular orbits. Measurements along this axis are referred to as along-track or transverse.
  - *W*-axis — Points normal to the orbital plane. Measurements along this axis are referred to as cross-track.

Version History

Introduced in R2020b

R2023a: Attitude Profile Block Update

Behavior changed in R2023a

To support idealized attitude control workflows, the Attitude Profile block has new parameters and an output port:

- **Output rotation from current to updated attitude** parameter
- **Output attitude** parameter
- $\mathbf{q}_{\mathbf{new}}$ port

See Also

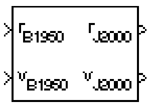
Attitude Dynamics | CubeSat Vehicle | Orbit Propagator | `juliandate`

Topics

“Model and Simulate CubeSats” on page 2-73

Besselian Epoch to Julian Epoch

Transform position and velocity components from discontinued Standard Besselian Epoch (B1950) to Standard Julian Epoch (J2000)



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Besselian Epoch to Julian Epoch block transforms two 3-by-1 vectors of Besselian Epoch position ($\bar{r}_{B1950}$), and Besselian Epoch velocity ($\bar{v}_{B1950}$) into Julian Epoch position ($\bar{r}_{J2000}$), and Julian Epoch velocity ($\bar{v}_{J2000}$). For more information on the transformation, see "Algorithms" on page 5-196.

Ports

Input

r_{B1950} — Position
3-by-1 vector

Position in Standard Besselian Epoch (B1950), specified as a 3-by-1 vector.

Data Types: `double`

v_{B1950} — Velocity
3-by-1 vector

Velocity in Standard Besselian Epoch (B1950), specified as a 3-by-1 vector.

Data Types: `double`

Output

r_{J2000} — Position
3-by-1 vector

Position in Standard Julian Epoch (J2000), returned as a 3-by-1 vector.

Data Types: `double`

v_{J2000} — Velocity
3-by-1 vector

Velocity in Standard Julian Epoch (J2000), returned as a 3-by-1 vector.

Data Types: `double`

Algorithms

The transformation is calculated using:

$$\begin{bmatrix} \bar{r}_{J2000} \\ \bar{v}_{J2000} \end{bmatrix} = \begin{bmatrix} \overline{M}_{rr} & \overline{M}_{vr} \\ \overline{M}_{rv} & \overline{M}_{vv} \end{bmatrix} \begin{bmatrix} \bar{r}_{B1950} \\ \bar{v}_{B1950} \end{bmatrix}$$

where $(\overline{M}_{rr}, \overline{M}_{vr}, \overline{M}_{rv}, M_{vv})$ are defined as:

$$\overline{M}_{rr} \begin{bmatrix} 0.9999256782 & -0.0111820611 & -0.0048579477 \\ 0.0111820610 & 0.9999374784 & -0.0000271765 \\ 0.0048579479 & -0.0000271474 & 0.9999881997 \end{bmatrix}$$

$$\overline{M}_{vr} = \begin{bmatrix} 0.00000242395018 & -0.00000002710663 & -0.00000001177656 \\ 0.00000002710663 & 0.00000242397878 & -0.00000000006587 \\ 0.00000001177656 & -0.00000000006582 & 0.00000242410173 \end{bmatrix}$$

$$\overline{M}_{rv} = \begin{bmatrix} -0.000551 & -0.238565 & 0.435739 \\ 0.238514 & -0.002667 & -0.008541 \\ -0.435623 & 0.012254 & 0.002117 \end{bmatrix}$$

$$\overline{M}_{vv} = \begin{bmatrix} 0.99994704 & -0.01118251 & -0.00485767 \\ 0.01118251 & 0.99995883 & -0.00002718 \\ 0.00485767 & -0.00002714 & 1.00000956 \end{bmatrix}$$

Version History

Introduced before R2006a

References

[1] "Supplement to Department of Defense World Geodetic System 1984 Technical Report: Part I - Methods, Techniques and Data Used in WGS84 Development," DMA TR8350.2-A.

Extended Capabilities

C/C++ Code Generation

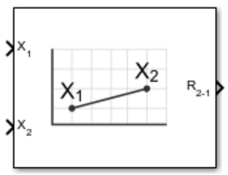
Generate C and C++ code using Simulink® Coder™.

See Also

Julian Epoch to Besselian Epoch

Calculate Range

Calculate range between two vehicles given their respective positions



Libraries:
Aerospace Blockset / GNC / Guidance

Description

The Calculate Range block computes the range between two vehicles. The equation used for the range calculation is

$$Range = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Ports

Input

$\mathbf{x_1}$ — Vehicle 1 position
3-element vector

Contains the (x, y, and z) position of vehicle 1, specified as a three-element vector. These values of the double data type.

Data Types: `double`

$\mathbf{x_2}$ — Vehicle 2 position
3-element vector

The (x, y, and z) position of vehicle 2, specified as a three-element vector. These values are of the double data type.

Data Types: `double`

Output

$\mathbf{R_{2\text{-}1}}$ — Range
scalar

Range from vehicle 2 and vehicle 1, returned as a scalar of double data type. The calculated range is the magnitude of the distance, but not the direction. It is always positive or zero.

Data Types: `double`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

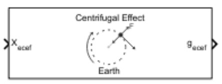
Generate C and C++ code using Simulink® Coder™.

See Also

Three-axis Inertial Measurement Unit

Centrifugal Effect Model

Implement mathematical representation of centrifugal effect for planetary gravity



Libraries:
Aerospace Blockset / Environment / Gravity

Description

The Centrifugal Effect Model block implements the mathematical representation of centrifugal effect for planetary gravity. The gravity centrifugal effect is the acceleration portion of centrifugal force effects due to the rotation of a planet. This block implements this representation using planetary rotation rates. You use centrifugal force values in rotating or non-inertial coordinate systems.

Ports

Input

X_{ecef} — Planet-centered planet-fixed coordinates
m-by-3 matrix

Planet-centered planet-fixed coordinates from the center of the planet, specified as a scalar. If **Planet model** has a value of `Earth`, this matrix contains Earth-centered Earth-fixed (ECEF) coordinates. The block does not use explicit units.

Data Types: `double`

ω — Planetary rotation rate
scalar

Planetary rotation rate, specified as a scalar, in rad/sec.

Dependencies

To enable this parameter, set **Planetary rotational rate (rad/sec)** to `Custom`.

Data Types: `double`

Output

Output 1 — Gravity values
m-by-3 array

Gravity values, returned as an m-by-3 array, in the *x*-axis, *y*-axis, and *z*-axis of the planet-centered planet-fixed coordinates, in input distance units per second squared.

Data Types: `double`

Parameters

Planet model — Planetary model

`Earth` (default) | `Venus` | `Mercury` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Custom`

Planetary model, specified as `Mercury`, `Venus`, `Earth`, `Moon`, `Mars`, `Jupiter`, `Saturn`, `Uranus`, `Neptune`, or `Custom`. The block uses the rotation of the selected planet to implement the mathematical representation of the centrifugal effect.

Dependencies

Selecting `Custom` enables the **Planetary rotational rate (rad/sec)** and **Input planetary rotation rate** parameters.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Mercury'` | `'Venus'` | `'Earth'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Custom'`
Default: `'Earth'`

Planetary rotational rate (rad/sec) — Planetary rotational rate

`7.2921150e-05` (default) | scalar

Planetary rotational rate in radians per second.

If you want to specify the planetary rotational rate as an input to the block, see the **Input planetary rotation rate** parameter.

Dependencies

Selecting the **Input planetary rotation rate** check box disables the **Planetary rotational rate (rad/sec)** parameter.

Programmatic Use
Block Parameter: `omega`
Type: character vector
Values: `'7.2921150e-05'` | scalar
Default: `'7.2921150e-05'`

Input planetary rotation rate — Planetary rotation rate port

`off` (default) | `on`

Select this check box to enable the ω input port. You can then input a planetary rotation rate as a block input.

Dependencies

Selecting this check box enables the ω and disables the **Planetary rotational rate (rad/sec)** parameter.

Programmatic Use
Block Parameter: `rate_loc`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Version History

Introduced in R2010a

References

[1] Vallado, David. *Fundamentals of Astrodynamics and Applications*. New York, NY: McGraw-Hill, 1997.

[2] "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems." NIMA TR8350.2.

Extended Capabilities

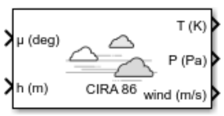
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Spherical Harmonic Gravity Model | Zonal Harmonic Gravity Model

CIRA-86 Atmosphere Model

Implement mathematical representation of 1986 CIRA atmosphere



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The CIRA-86 Atmosphere Model block implements the mathematical representation of the 1986 Committee on Space Research (COSPAR) International Reference Atmosphere (CIRA). The block provides values for mean temperature, pressure, and zonal wind speed for the input geopotential altitude.

The CIRA-86 Atmosphere Model block port labels change based on the input and output units selected from the **Units** list.

Limitations

- This block uses a corrected version of the CIRA data files provided by J. Barnett in July 1990 in ASCII format.
- This block has the limitations of the CIRA 1986 model.
- The CIRA 1986 model that CIRA-86 Atmosphere Model implements limits values to latitudes of 80 degrees S to 80 degrees N on Earth and geopotential heights of 0 to 120 kilometers. The CIRA-86 Atmosphere Model block clips output beyond these ranges.

  The CIRA 1986 model data also has missing data points.

  - In each monthly mean data set, the model omits values at 80 degrees S for 101,300 pascal or 0 meters, because these levels are within the Antarctic land mass.
  - In the annual mean data set for latitudes 70, 75, and 80 degrees S, the model also omits these values:

    - 101,300 and 78,895 pascal
    - 0 and 2,500 meters

  As a result of the missing data points, input for pressures above 61,441 pascal or geopotential heights below 5000 meters at these latitudes generate `NaN` output.

  For zonal mean pressure in constant altitude coordinates, pressure data is not available below 20 km. This value is the bottom level of the CIRA climatology.

  When the CIRA-86 Atmosphere Model block detects an out-of-range input, the block uses the **Action for out-of-range input** parameter to determine the block behavior.

Ports

Input

Port_1 — Latitude
array

Contains the latitude in degrees (limited to +/-80 degrees).

Data Types: `double`

Port_2 — Geopotential heights or pressures
array

Contains an `m` array of either:

- Geopotential heights in selected length units (**Coordinate type** is `GPHeight`)
- Pressures in selected pressure units (**Coordinate type** is `Pressure`)

Data Types: `double`

Output

Port_1 — Mean temperature
array

Mean temperature, specified as an array, in selected units.

Data Types: `double`

Port_2 — Pressures or geopotential heights
array

`m` array of either:

- Pressures in selected pressure units (**Coordinate type** is `GPHeight`)
- Geopotential heights in selected length units (**Coordinate type** is `Pressure`)

Data Types: `double`

Port_3 — Mean zonal winds
array

Mean zonal winds, specified as an array, in selected units.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Height	Temperature	Speed of Sound	Air Pressure	Air Density
`Metric (MKS)`	Meters	Kelvin	Meters per second	Pascal	Kilograms per cubic meter
`English (Velocity in ft/s)`	Feet	Degrees Rankine	Feet per second	Pound-force per square inch	Slug per cubic foot
`English (Velocity in kts)`	Feet	Degrees Rankine	Knots	Pound-force per square inch	Slug per cubic foot

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Coordinate type — Coordinate type representation

`Pressure` (default) | `GPHeight`

Coordinate type representation, specified as:

- `Pressure`

  Indicates pressure in pascal.
- `GPHeight`

  Indicates geopotential height in meters.

Programmatic Use
Block Parameter: `ctype`
Type: character vector
Values: `'GPHeight'` | `'Pressure'`
Default: `'GPHeight'`

Mean value type — Mean value types

`Monthly` (default) | `Annual`

Mean value types, specified as:

- `Monthly`

  Indicates monthly values. If you select `Monthly`, you must also set the **Month** parameter.
- `Annual`

  Indicates annual values. Valid when **Coordinate type** has a value of `Pressure`.

Dependencies

Setting this parameter to `Monthly` enables the **Month** parameter.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `'Monthly'` | `'Annual'`
Default: `'Monthly'`

Month — Month of mean value

`January` (default) | `February` | `March` | `April` | `May` | `June` | `July` | `August` | `September` | `October` | `November` | `December`

Month in which the mean values are taken.

Dependencies

This parameter is enabled when **Mean value type** is set to `Monthly`.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced in R2007b

R2023b: CIRA-86 Atmosphere Model Block Update

The CIRA-86 Atmosphere Model block has been updated to take action when inputs to the function are outside the CIRA 1986 model limitations (see “Limitations” on page 5-203).

References

[1] Fleming, E. L., Chandra, S., Shoeberl, M. R., Barnett, J. J., *Monthly Mean Global Climatology of Temperature, Wind, Geopotential Height and Pressure for 0-120 km*, NASA TM100697, February 1988.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

COESA Atmosphere Model | ISA Atmosphere Model

External Websites

https://ccmc.gsfc.nasa.gov/

Climb Rate Indicator

Display measurements for aircraft climb rate

Libraries:
Aerospace Blockset / Flight Instruments

Description

The Climb Rate Indicator block displays measurements for an aircraft climb rate in ft/min.

The needle covers the top semicircle, if the velocity is positive, and the lower semicircle, if the climb rate is negative. The range of the indicator is from –**Maximum** feet per minute to **Maximum** feet per minute. Major ticks indicate **Maximum**/4. Minor ticks indicate **Maximum**/8 and **Maximum**/80.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name

Connect to signal for display, selected from list of signal names.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Maximum — Maximum tick mark value

4000 (default) | finite | double | scalar

Maximum tick mark value, specified as a finite double scalar value, in ft/min.

The minimum tick value is always `0`.

Programmatic Use
Block Parameter: `MaximumRate`
Type: character vector
Values: scalar

Default: `'4000'`

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is ignored for code generation.

See Also

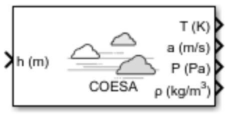
Airspeed Indicator | Altimeter | Artificial Horizon | Exhaust Gas Temperature (EGT) Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Topics

"Display Measurements with Cockpit Instruments" on page 2-59
"Programmatically Interact with Gauge Band Colors" on page 2-61
"Flight Instrument Gauges" on page 2-58

COESA Atmosphere Model

Implement 1976 COESA lower atmosphere



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The COESA Atmosphere Model block implements the mathematical representation of the 1976 Committee on Extension to the Standard Atmosphere (COESA) United States standard lower atmospheric values for absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.

The COESA Atmosphere Model, Non-Standard Day 210C, and Non-Standard Day 310 blocks are identical blocks. When configured for COESA Atmosphere Model, the block implements the COESA mathematical representation. When configured for Non-Standard Day 210C, the block implements MIL-STD-210C climatic data. When configured for Non-Standard Day 310, the block implements MIL-HDBK-310 climatic data.

Below 32,000 meters (approximately 104,987 feet), the U.S. Standard Atmosphere is identical with the Standard Atmosphere of the International Civil Aviation Organization (ICAO).

The COESA Atmosphere Model block port labels change based on the input and output units selected from the **Units** list.

Limitations

Below the geopotential altitude of 0 m (0 feet) and above the geopotential altitude of 84,852 m (approximately 278,386 feet), temperature values are extrapolated linearly and pressure values are extrapolated logarithmically. Density and speed of sound are calculated using a perfect gas relationship.

Ports

Input

Port_1 — Geopotential height
scalar | array

Geopotential height, specified as a scalar or array, in specified units.

Data Types: `double`

Output

Port_1 — Temperature
scalar | array

Temperature, specified as a scalar or array, in specified units.

Data Types: `double`

Port_2 — Speed of sound
scalar | array

Speed of sound, specified as a scalar or array, in specified units.

Data Types: `double`

Port_3 — Air pressure
scalar | array

Air pressure, specified as a scalar or array, in specified units.

Data Types: `double`

Port_4 — Air density
scalar | array

Air density, specified as a scalar or array, in specified units.

Data Types: `double`

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Height	Temperature	Speed of Sound	Air Pressure	Air Density
`Metric (MKS)`	Meters	Kelvin	Meters per second	Pascal	Kilograms per cubic meter
`English (Velocity in ft/s)`	Feet	Degrees Rankine	Feet per second	Pound-force per square inch	Slug per cubic foot
`English (Velocity in kts)`	Feet	Degrees Rankine	Knots	Pound-force per square inch	Slug per cubic foot

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Specification — Atmosphere model type

`1976 COESA-extended U.S. Standard Atmosphere` (default) | `MIL-HDBK-310` | `MIL-STD-210C`

Atmosphere model type, specified as `1976 COESA-extended U.S. Standard Atmosphere`, `MIL-HDBK-310`, or `MIL-STD-210C`. For the `MIL-HDBK-310` and `MIL-STD-210C` options:

`MIL-HDBK-310`	This selection is linked to the Non-Standard Day 310 block. See the block reference for more information. Selecting `MIL-HDBK-310` enables the parameters **Atmospheric model type**, **Extreme parameter**, **Frequency of occurrence**, and **Altitude of extreme value**.
`MIL-STD-210C`	This selection is linked to the Non-Standard Day 210C block. See the block reference for more information. Selecting `MIL-HDBK-310` enables the parameters **Atmospheric model type**, **Extreme parameter**, **Frequency of occurrence**, and **Altitude of extreme value**.

Dependencies

Selecting `MIL-HDBK-310` or `MIL-STD-210C` enables these parameters:

- **Atmospheric model type**
- **Extreme parameter**
- **Frequency of occurrence**
- **Altitude of extreme value**

Programmatic Use

Block Parameter: `spec`
Type: character vector
Values: `'1976 COESA-extended U.S. Standard Atmosphere'` | `'MIL-HDBK-310'` | `'MIL-STD-210C'`
Default: `'1976 COESA-extended U.S. Standard Atmosphere'`

Atmospheric model type — Model type

`Profile` (default) | `Envelope`

Representation of atmospheric model type, specified as:

`Profile`	Realistic atmospheric profiles associated with extremes at specified altitudes. Recommended for simulation of vehicles vertically traversing the atmosphere or when the total influence of the atmosphere is needed.
`Envelope`	Uses extreme atmospheric values at each altitude. Recommended for vehicles only horizontally traversing the atmosphere without much change in altitude.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- Selecting `Profile` enables the **Attitude of extreme value** parameter.

Programmatic Use

Block Parameter: `model`

Type: character vector
Values: `'Profile'` | `'Envelope'`
Default: `'Profile'`

Extreme parameter — Model type

`High temperature` (default) | `Low temperature` | `High density` | `Low density` | `High pressure` | `Low pressure`

Atmospheric parameter that is the extreme value.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- The `High pressure` and `Low pressure` options appear only when **Atmospheric model type** is set to `Envelope`.

Programmatic Use
Block Parameter: `profile_var`
Type: character vector
Values: `'High temperature'` | `'Low temperature'` | `'High density'` | `'Low density'` | `'High pressure'` | `'Low pressure'`
Default: `'High temperature'`

Frequency of occurrence — Model type

`1%` (default) | `Extreme values` | `5%` | `10%` | `20%`

Percent of time the values would occur.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- `Extreme values`, `5%`, and `20%` are available only when `Envelope` is selected for **Atmospheric model type**.
- `1%` and `10%` are always available.

Programmatic Use
Block Parameter: `profile_percent`
Type: character vector
Values: `'Extreme values'` | `'1%'` | `'5%'` | `'10%'` | `'20%'`
Default: `'1%'`

Altitude of extreme value — Geometric altitude

`5 km (16404 ft)` (default) | `10 km (32808 ft)` | `20 km (65617 ft)` | `30 km (98425 ft)` | `40 km (131234 ft)`

Geometric altitude at which the extreme values occur, specified as `5 km (16404 ft)`, `10 km (32808 ft)`, `20 km (65617 ft)`, `30 km (98425 ft)`, or `40 km (131234 ft)`.

Dependencies

This parameter appears if the **Atmospheric model type** is set to `Profile`.

Programmatic Use
Block Parameter: `profile_alt`
Type: character vector
Values: `5 km (16404 ft)` | `10 km (32808 ft)` | `20 km (65617 ft)` | `30 km (98425 ft)` | `40 km (131234 ft)`
Default: `40 km (131234 ft)`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History
Introduced before R2006a

References

[1] *U.S. Standard Atmosphere.*, Washington, D.C.: U.S. Government Printing Office, 1976.

Extended Capabilities

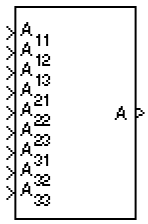
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
CIRA-86 Atmosphere Model | ISA Atmosphere Model | Non-Standard Day 210C | Non-Standard Day 310

Topics
“NASA HL-20 Lifting Body Airframe” on page 3-14

Create 3x3 Matrix

Create 3-by-3 matrix from nine input values



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Create 3x3 Matrix block creates a 3-by-3 matrix from nine input values where each input corresponds to an element of the matrix.

The output matrix has the form of

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Ports

Input

A_{11} — First row, first column of matrix
matrix element

First row, first column of the matrix, specified as a matrix element.

Example: 1

Data Types: `double`

A_{12} — First row, second column of matrix
matrix element

First row, second column of the matrix, specified as a matrix element.

Example: 2

Data Types: `double`

A_{13} — First row, third column of matrix
matrix element

First row, third column of the matrix, specified as a matrix element.

Example: 3

Data Types: `double`

A_{21} — Second row, first column of matrix
matrix element

Second row, first column of the matrix, specified as a matrix element.

Example: 4

Data Types: `double`

A_{22} — Second row, second column of matrix
matrix element

Second row, second column of the matrix, specified as a matrix element.

Example: 5

Data Types: `double`

A_{23} — Second row, third column of matrix
matrix element

Second row, third column of the matrix, specified as a matrix element.

Example: 6

Data Types: `double`

A_{31} — Third row, first column of matrix
matrix element

Third row, first column of the matrix, specified as a matrix element.

Example: 7

Data Types: `double`

A_{32} — Third row, second column of matrix
matrix element

Third row, second column of the matrix, specified as a matrix element.

Example: 8

Data Types: `double`

A_{33} — Third row, third column of matrix
matrix element

Third row, third column of the matrix, specified as a matrix element.

Example: 9

Data Types: `double`

Output

A — Matrix
3-by-3 matrix

Matrix, output as a 3-by-3 matrix.

Data Types: `double`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

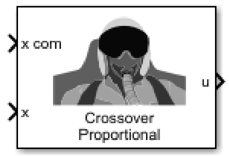
Generate C and C++ code using Simulink® Coder™.

See Also

Adjoint of 3x3 Matrix | Determinant of 3x3 Matrix | Invert 3x3 Matrix | Symmetric Inertia Tensor

Crossover Pilot Model

Represent crossover pilot model



Libraries:
Aerospace Blockset / Pilot Models

Description

The Crossover Pilot Model block represents the pilot model described in *Mathematical Models of Human Pilot Behavior* [1]). This pilot model is a single input, single output (SISO) model that represents some aspects of human behavior when controlling aircraft.

The Crossover Pilot Model takes into account the combined dynamics of the human pilot and the aircraft, using the form described in "Algorithms" on page 5-222 around the crossover frequency.

This block has nonlinear behavior. If you want to linearize the block (for example, with one of the `linmod` functions), you might need to change the Pade approximation order. The Crossover Pilot Model block implementation incorporates the Transport Delay block with the **Pade order (for linearization)** parameter set to `2` by default. To change this value, use the `set_param` function, for example:

```
set_param(gcb,'pade','3')
```

When modeling human pilot models, use this block for more accuracy than that provided by the Tustin Pilot Model block. This block is also less accurate than the Precision Pilot Model block.

Ports

Input

x com — Signal command
scalar

Signal command that the pilot model controls, specified as a scalar.

Data Types: `double`

x — Signal
scalar

Signal that the pilot model controls, specified as a scalar.

Data Types: `double`

Output

u — Aircraft command
scalar

Aircraft command, returned as a scalar.

Data Types: `double`

Parameters

Type of control — Dynamics control

`Proportional` (default) | `Rate or velocity` | `Spiral divergence` | `Second order - Short period` | `Acceleration(*)` | `Roll attitude(*)` | `Unstable short period(*)` | `Second order - Phugoid(*)`

Dynamics control that you want the pilot to have over the aircraft. This table lists the options and associated dynamics.

Option (Controlled Element Transfer Function)	Transfer Function of Controlled Element (Y_c)	Transfer Function of Pilot (Y_p)	$Y_c Y_p$	Notes
`Proportional`	K_c	$\frac{K_p e^{-\tau s}}{s}$	$\frac{K_c K_p e^{-\tau s}}{s}$	
`Rate or velocity`	$\frac{K_c}{s}$	$K_p e^{-\tau s}$	$\frac{K_c K_p e^{-\tau s}}{s}$	
`Spiral divergence`	$\frac{K_c}{T_I s - 1}$	$K_p e^{-\tau s}$	$\frac{K_c K_p e^{-\tau s}}{(T_I s - 1)}$	
`Second order - Short period`	$\frac{K_c \omega_n{}^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$	$\frac{K_p e^{-\tau s}}{T_I s + 1}$	$\frac{K_c \omega_n{}^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \times$ $\frac{K_p e^{-\tau s}}{T_I s + 1}$	Short period, with $\omega_n > 1/\tau$
`Acceleration (*)`	$\frac{K_c}{s^2}$	$K_p s e^{-\tau s}$	$\frac{K_c K_p e^{-\tau s}}{s}$	
`Roll attitude (*)`	$\frac{K_c}{s(T_I s + 1)}$	$K_p(T_L s + 1)e^{-\tau s}$	$\frac{K_c K_p e^{-\tau s}}{s}$	With $T_L \approx T_I$
`Unstable short period(*)`	$\frac{K_c}{(T_{I1} s + 1)(T_{I2} s - 1)}$	$K_p(T_L s + 1)e^{-\tau s}$	$\frac{K_c K_p e^{-\tau s}}{(T_{I2} s - 1)}$	With $T_L \approx T_{I1}$
`Second order - Phugoid(*)`	$\frac{K_c \omega_n{}^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$	$K_p(T_L s + 1)e^{-\tau s}$	$\frac{K_c K_p \omega_n^2 e^{-\tau s}}{s}$	Phugoid, with $\omega_n \ll 1/\tau$, $1/T_L \approx \zeta\omega_n$

* Indicates that the pilot model includes a Derivative block, which produces a numerical derivative. For this reason, do not send discontinuous (such as a step) or noisy input to the Crossover Pilot Model block. Such inputs can cause large outputs that might render the system unstable.

This table defines the variables used in the list of control options.

Variable	Description
K_c	Aircraft gain.
K_p	Pilot gain.
τ	Pilot time delay.
T_I	Lag constant.
T_L	Lead constant.
ζ	Damping ratio for the aircraft.
ω_n	Natural frequency of the aircraft.

Dependencies

The Crossover Pilot Model parameters are enabled and disabled according to the **Type of control** options. The **Calculated value**, **Controlled element gain**, **Pilot gain**, **Crossover frequency (rad/s)**, and **Pilot time delay(s)** parameters are always enabled.

Programmatic Use
Block Parameter: `sw_popup`
Type: character vector
Values: `'Proportion'` | `'Rate or velocity'` |`'Spiral divergence'` | `'Second order - Short period'` | `'Acceleration(*)'` | `'Roll attitude(*)'` | `'Unstable short period(*)'` | `'Second order - Phugoid(*)'`
Default: `'Proportion'`

Calculated value — Crossover frequency or pilot gain

`Crossover frequency` (default) | `Pilot gain`

Crossover frequency or pilot gain value you want the block to calculate:

- `Crossover frequency` — The block calculates the crossover frequency value. The parameter value is disabled.
- `Pilot gain` — The block calculates the pilot gain value. The parameter value is disabled.

Programmatic Use
Block Parameter: `freq_gain_popup`
Type: character vector
Values: `'Crossover frequency'` | `'Pilot gain'`
Default: `'Crossover frequency'`

Controlled element gain — Controlled element gain

1 (default) | scalar

Controlled element gain, specified as a double scalar.

Programmatic Use
Block Parameter: `Kc`
Type: character vector
Values: double scalar
Default: `'1'`

Pilot gain — Pilot gain

3 (default) | scalar

Pilot gain, specified as a double scalar.

Dependencies

To enable this parameter, set **Calculated value** to `Pilot gain`.

Programmatic Use
Block Parameter: Kp
Type: character vector
Values: double scalar
Default: '3'

Crossover frequency (rad/s) — Crossover frequency

3 (default) | scalar in the range of 1 and 10

Crossover frequency value, specified as double scalar, in rad/s. The value must be in the range between 1 and 10.

Dependencies

To enable this parameter, set **Calculated value** to `Crossover frequency`.

Programmatic Use
Block Parameter: `omega_c`
Type: character vector
Values: double scalar
Default: '3'

Pilot time delay(s) — Pilot time delay

`0.1` (default) | scalar

Total pilot time delay, specified as a double scalar, in seconds. This value typically ranges from 0.1 s to 0.2 s.

Programmatic Use
Block Parameter: `time_delay`
Type: character vector
Values: double scalar
Default: '0.1'

Pilot lead constant — Pilot lead constant

1 (default) | scalar

Pilot lead constant, specified as a double scalar.

Dependencies

To enable this parameter, set **Type of control** to one of the following options:

- `Roll attitude (*)`

- `Unstable short period (*)`
- `Second order - Phygoid(*)`

Programmatic Use
Block Parameter: `T`
Type: character vector
Values: double scalar
Default: `'1'`

Pilot lag constant — Pilot lag constant

5 (default) | scalar

Pilot lag constant, specified as a double scalar.

Dependencies

To enable this parameter, set **Type of control** to `Second order - Short period`.

Programmatic Use
Block Parameter: `Ti`
Type: character vector
Values: double scalar
Default: `'5'`

Algorithms

The Crossover Model takes into account the combined dynamics of the human pilot and the aircraft, using the following form around the crossover frequency:

$$Y_p Y_c = \frac{\omega_c e^{-\tau s}}{s},$$

Where:

Variable	Description
Y_p	Pilot transfer function.
Y_c	Aircraft transfer function.
ω_c	Crossover frequency.
τ	Transport delay time caused by the pilot neuromuscular system.

If the dynamics of the aircraft (Y_c) change, Y_p changes correspondingly.

Note This block is valid only around the crossover frequency. It is not valid for discrete inputs such as a step.

Version History

Introduced in R2012b

References

[1] McRuer, D. T., Krendel, E., *Mathematical Models of Human Pilot Behavior*. Advisory Group on Aerospace Research and Development AGARDograph 188, Jan. 1974.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

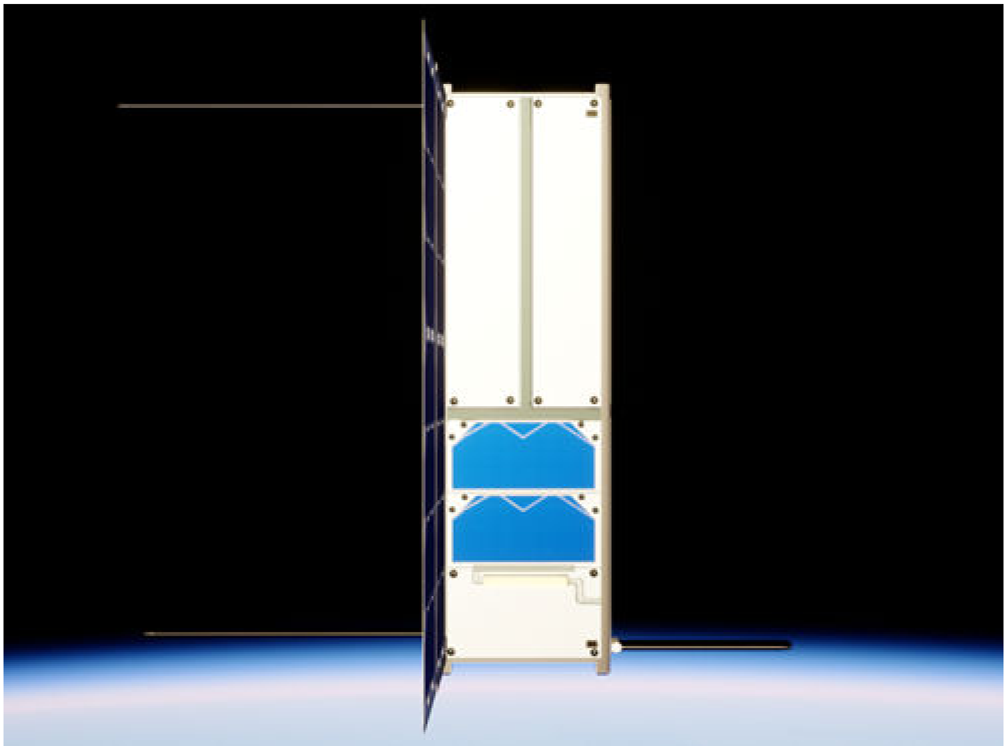
See Also

Precision Pilot Model | Tustin Pilot Model | Transport Delay | `linmod`

CubeSat

Generic CubeSat

Description



CubeSat is one of the spacecraft that you can use within the 3D simulation environment. The Aerospace Blockset supports a CubeSat that represents a 3U-sized satellite, with the origin at its center. This environment is rendered using the Unreal Engine from Epic Games. For detailed views of the CubeSat, see “Views” on page 5-477.

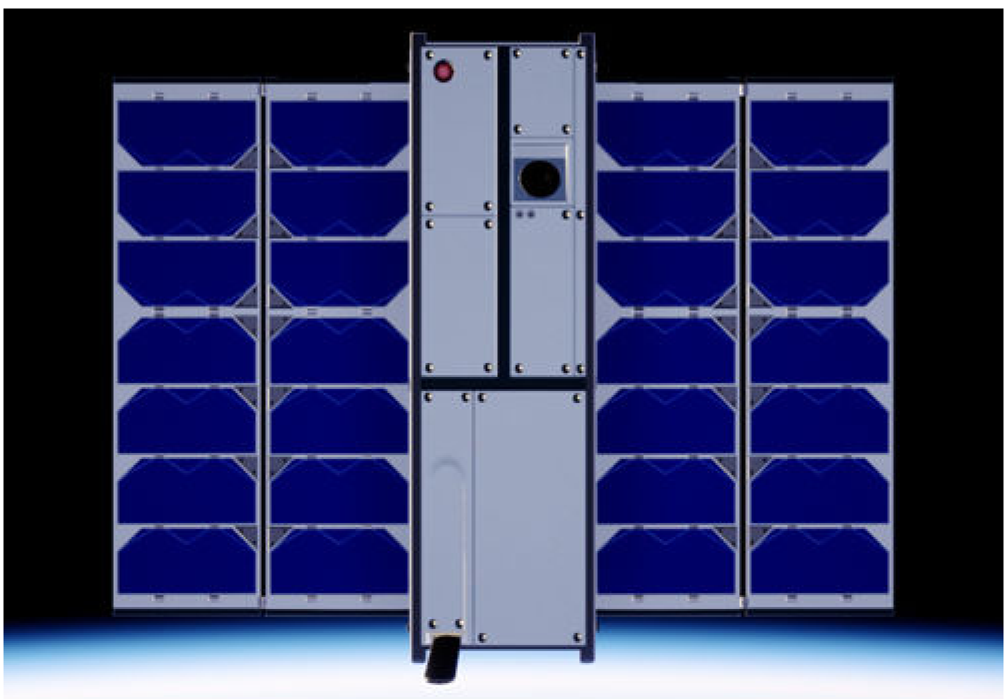
To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Spacecraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Parameters** tab, set the **Type** parameter to `CubeSat`.
3. Set the **Initial translation (m)** and **Initial rotation (rad)** parameters to an array size that matches the CubeSat spacecraft, for example, `zeros(10,3)`.

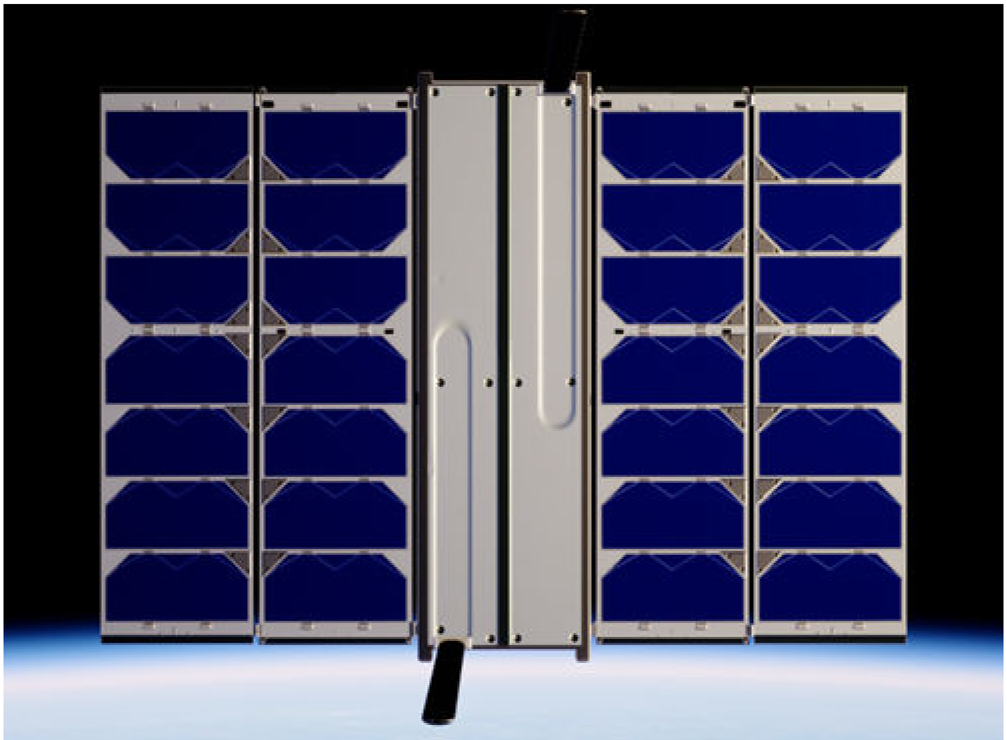
Views

`Top-down view` — CubeSat top-down view
diagram

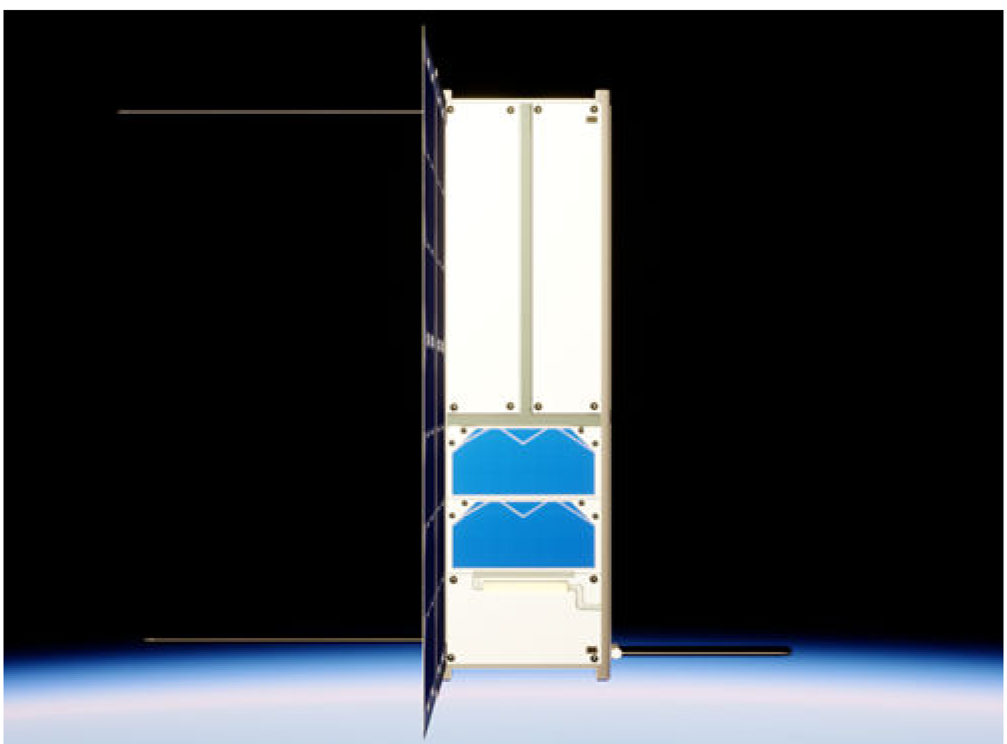
Left side view — CubeSat left side view
diagram



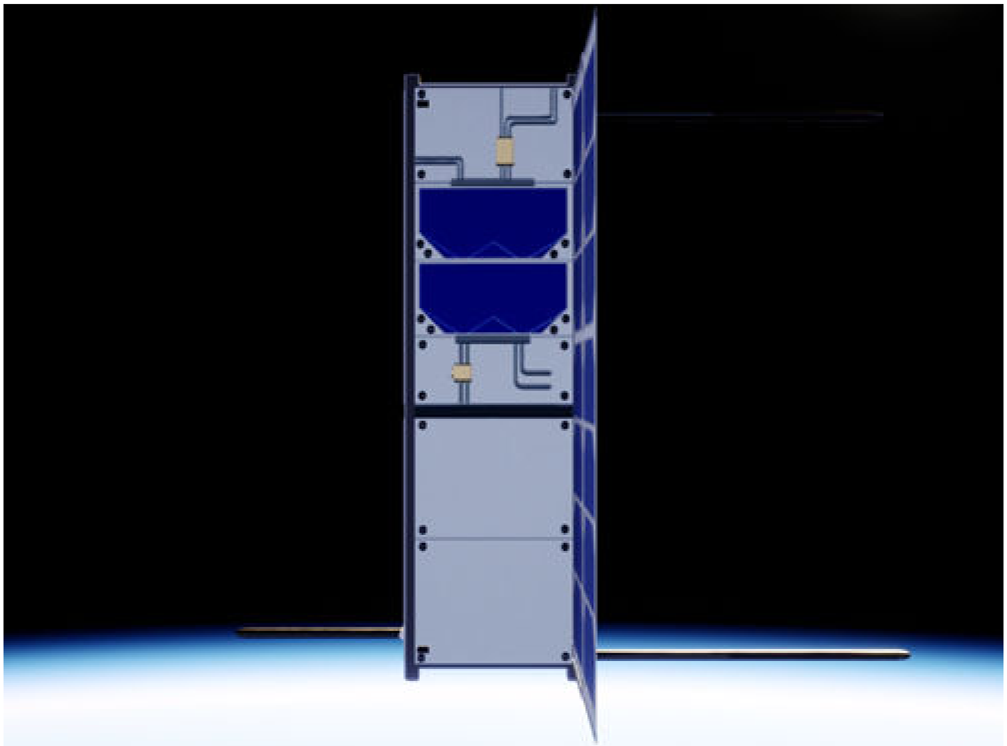
Right side view — CubeSat right side view
diagram



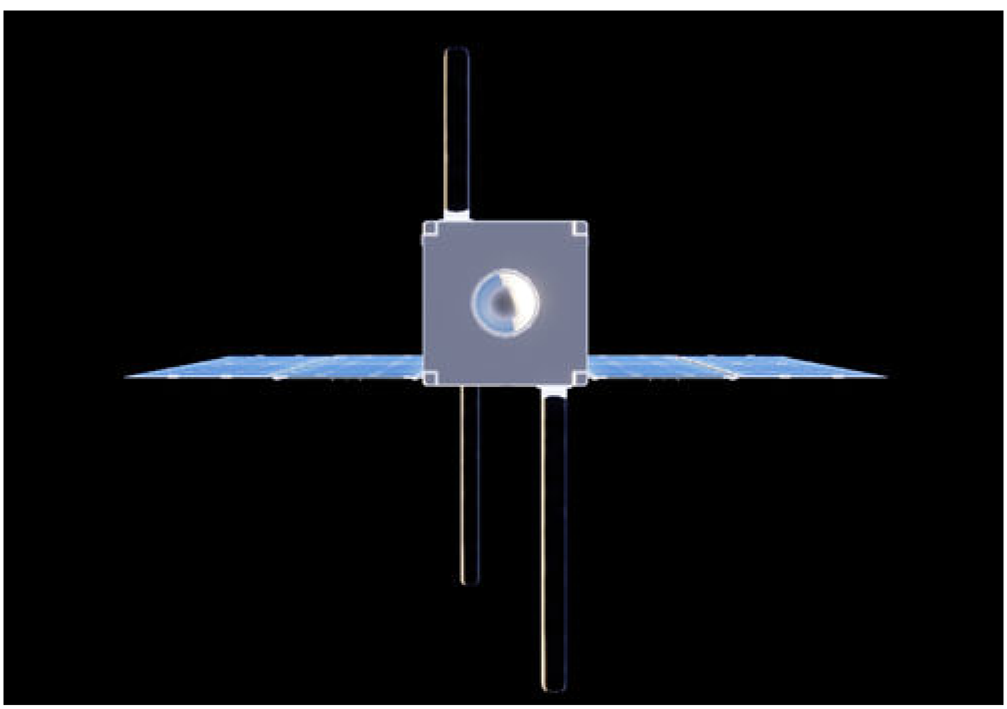
`Front view` — CubeSat front view
diagram



`Back view` — CubeSat back view
diagram



`Bottom view` — CubeSat bottom view
diagram



Skeleton

CubeSat Skeleton

The CubeSat skeleton has these parts.

- `CubeSat`
  - `ANTENNA1`
  - `ANTENNA2`
  - `ANTENNA3`
  - `SENSOR`
  - `SOLAR_ARRAY1`
    - `SOLAR_ARRAY1_1`
  - `SOLAR_ARRAY2`
    - `SOLAR_ARRAY2_2`
  - `THRUSTER`

Version History

Introduced in R2024a

See Also

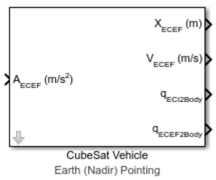
Blocks
Simulation 3D Spacecraft

Tools
Simulation 3D CubeSat Pack

Topics
“Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37
“Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2
“How 3D Simulation for Aerospace Blockset Works” on page 2-40

CubeSat Vehicle

Model CubeSat vehicle



Libraries:
Aerospace Blockset / Spacecraft / CubeSat Vehicles

Description

The CubeSat Vehicle block models CubeSat vehicles to provide a high level mission planning/rapid prototyping option to quickly model and propagate satellite orbits, one satellite at a time. (To propagate multiple satellites simultaneously, see the Orbit Propagator block.) To accommodate constellation planning workflows, you can also use these blocks multiple times in a model. Specify this information for the vehicle:

- Initial orbital state
- Attitude control (pointing) mode

The library contains three versions of the CubeSat Vehicle block preconfigured for these common attitude control modes:

- Earth (Nadir) Pointing — Primary alignment vector points towards the center of the Earth
- Sun Tracking — Primary alignment vector points toward the Sun
- Custom Pointing — Custom alignment and constraint vectors

Ports

Input

A_{ECEF} (m/s^2) — Vehicle accelerations
vector of size 3

Vehicle gravity accelerations (including gravity) used for orbit propagation, specified as a vector of size 3, in m/s^2.

Data Types: `single` | `double`

1st Alignment$_{Body}$ — Primary alignment vector
three-element vector

Primary alignment vector, in the Body frame, to align with primary constraint vector.

Data Types: `double`

1st Constraint$_{ECI}$ — Primary constraint vector
three-element vector

Primary constraint vector specifying the direction in which to align the primary alignment vector.

Dependencies

This port is not available when **Pointing mode** is set to `Earth (Nadir) Pointing` or `Sun Tracking`, which have implied primary constraint vectors.

Data Types: `double`

1st Alignment$_{Body}$ — Primary alignment vector
three-element vector

Primary alignment vector, in the Body frame, to align with primary constraint vector.

Data Types: `double`

1st Constraint$_{ECI}$ — Primary constraint vector
three-element vector

Primary constraint vector specifying the direction in which to align the primary alignment vector.

Dependencies

- The direction depends on the **Constraint coordinate system**.
- This port is not available when **Pointing mode** is set to `Earth (Nadir) Pointing` or `Sun Tracking`, which have implied primary constraint vectors.

Data Types: `double`

2nd Alignment$_{Body}$ — Secondary alignment vector
three-element vector

Secondary alignment vector, in the Body frame, to align with secondary constraint vector.

Data Types: `double`

2nd Constraint$_{ECI}$ — Secondary constraint vector
three-element vector

Secondary constraint vector specifying the direction in which to align the secondary alignment vector.

Dependencies

The direction depends on the **Constraint coordinate system**.

Data Types: `double`

Output

X$_{ECEF}$ — CubeSat position
three-element vector

Earth-centered Earth-fixed CubeSat position components, specified as a 3-by-1 array.

Data Types: `double`

V$_{ECEF}$ — Velocity components
3-by-1 array

Earth-centered Earth-fixed velocity components, specified as a 3-by-1 array.

Data Types: `double`

$q_{ECI2Body}$ — Quaternion rotation
4-by-1 array

Quaternion rotation from Earth-centered inertial frame to Body frame.

Data Types: `double`

$q_{ECEF2Body}$ — Quaternion array
4-by-1 array

Quaternion rotation from Earth-centered Earth-fixed frame to Body frame.

Data Types: `double`

Parameters

Start date [Julian date] — Initial start date of simulation

`2458488` (default) | Julian date

Initial start date of simulation. The block defines initial conditions using this date.

Tip To calculate the Julian date, use the `juliandate` function.

Programmatic Use
Block Parameter: `sim_t0`
Type: character vector
Values: Julian date
Default: `'2458488'`

CubeSat Orbit

Input method — Initial vehicle

`Keplerian Orbital Elements` (default) | `ECI Position and Velocity` | `ECEF Position and Velocity` | `Geodetic LatLonAlt and Velocity in NED`

Initial vehicle position and velocity input method.

Dependencies

Selecting the `Keplerian Orbital Elements` input method enables these parameters:

- **Epoch of ECI frame [Julian date]**
- **Semi-major axis [m]**
- **Eccentricity**
- **Inclination [deg]**
- **Right ascension of the ascending node [deg]**

- **Argument of periapsis [deg]**
- **True anomaly [deg]**
- **True longitude [deg] (circular equatorial)**
- **Argument of latitude [deg] (circular inclined)**
- **Longitude of periapsis [deg] (elliptical equatorial)**

Selecting the `ECI Position and Velocity` input method enables these parameters:

- **Epoch of ECI frame [Julian date]**
- **ECI position vector [m]**
- **ECI velocity vector [m/s]**

Selecting the `ECEF Position and Velocity` input method enables these parameters:

- **ECEF position vector [m]**
- **ECEF velocity vector [m/s]**

Selecting the `Geodetic LatLonAlt and Velocity in NED` input method enables these parameters:

- **Geodetic latitude, longitude, altitude [deg, deg, m]**
- **NED velocity vector [m/s]**

Programmatic Use
Block Parameter: `method`
Type: character vector
Values: `'Keplerian Orbital Elements'` | `'ECI Position and Velocit'` | `'ECEF Position and Velocity'` | `'Geodetic LatLonAlt and Velocity in NED'`
Default: `'Keplerian Orbital Elements'`

Epoch of ECI frame [Julian date] — Epoch of ECI frame

`2451545` (default) | Julian date

Epoch of ECI frame, specified as a Julian date.

Tip To calculate the Julian date for a particular date, use the `juliandate` function.

Programmatic Use
Block Parameter: `epoch`
Type: character vector
Values: Julian date format
Default: `'2451545'`

Semi-major axis [m] — CubeSat semi-major axis

`6878137` (default) | axis in meters

CubeSat semi-major axis (half of the longest orbit diameter), specified in m.

Programmatic Use
Block Parameter: `a`
Type: character vector
Values: scalar
Default: `'6878137'`

Eccentricity — Orbital eccentricity

`0` (default) | eccentricity greater than or equal to 0

Deviation of the CubeSat orbit from a perfect circle.

Programmatic Use
Block Parameter: `ecc`
Type: character vector
Values: scalar
Default: `'0'`

Inclination [deg] — Tilt angle of CubeSat orbital plane

`0` | degrees between 0 and 180

Tilt angle of CubeSat orbital plane, specified between 0 and 180 deg.

Programmatic Use
Block Parameter: `incl`
Type: character vector
Values: scalar
Default: `'0'`

Right ascension of the ascending node [deg] — Angular distance in equatorial plane

`0` (default) | degrees between 0 and 360

Angular distance in equatorial plane from *x*-axis to location of the ascending node (point at which the satellite crosses the equator from south to north), specified between 0 and 360 deg.

Programmatic Use
Block Parameter: `omega`
Type: character vector
Values: scalar
Default: `'0'`

Argument of periapsis [deg] — Angle from CubeSat body ascending node to periapsis

`0` (default) | degrees between 0 and 360

Angle from the CubeSat body ascending node to the periapsis (closest point of orbit to Earth), specified between 0 and 360 deg.

Programmatic Use
Block Parameter: `argp`
Type: character vector
Values: scalar
Default: `'0'`

True anomaly [deg] — Angle between periapsis and current position of CubeSat

`0` (default) | degrees between 0 and 360

Angle between the periapsis (closest point of orbit to Earth) and the current position of CubeSat, specified between 0 and 360 deg.

Programmatic Use
Block Parameter: `nu`
Type: character vector
Values: scalar
Default: `'0'`

True longitude [deg] (circular equatorial) — Angle between *x*-axis of periapsis and position of CubeSat vector

`0` (default) | degrees between 0 and 360

Angle between *x*-axis of periapsis and position of CubeSat vector, specified between 0 and 360 deg.

Programmatic Use
Block Parameter: `truelon`
Type: character vector
Values: scalar
Default: `'0'`

Argument of latitude [deg] (circular inclined) — Angle between ascending node and satellite position vector

`0` (default) | degrees between 0 and 360

Angle between ascending node and satellite position vector, specified between 0 and 360 deg.

Programmatic Use
Block Parameter: `arglat`
Type: character vector
Values: scalar
Default: `'0'`

Longitude of periapsis [deg] (elliptical equatorial) — Angle between *x*-axis of periapsis and eccentricity vector

`0` (default) | degrees between 0 and 360

Angle between the *x*-axis of the periapsis and the eccentricity vector, specified between 0 and 360 deg.

Programmatic Use
Block Parameter: `lonper`
Type: character vector
Values: scalar
Default: `'0'`

ECI position vector [m] — Cartesian position vector

`[0 0 0]` (default) | vector

Cartesian position vector of satellite in ECI coordinate frame at **Start Date**.

Programmatic Use
Block Parameter: `r_eci`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

ECI velocity vector [m/s] — Cartesian velocity vector

`[0 0 0]` (default) | velocity vector

Cartesian velocity vector of satellite in ECI coordinate frame at **Start Date**.

Programmatic Use
Block Parameter: `v_eci`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

ECEF position vector [m] — Cartesian position vector

`[0 0 0]` (default) | vector

Cartesian position vector of satellite in ECEF coordinate frame at **Start Date**.

Programmatic Use
Block Parameter: `r_ecef`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

ECEF velocity vector [m/s] — Cartesian velocity vector

`[0 0 0]` (default) | velocity vector

Cartesian velocity vector of satellite in ECEF coordinate frame at **Start Date**.

Programmatic Use
Block Parameter: `v_ecef`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

Geodetic latitude, longitude, altitude [deg, deg, m] — Geodetic latitude and longitude, and altitude

`[0 0 0]` (default) | velocity vector

Geodetic latitude and longitude, in deg, and altitude above WGS84 ellipsoid, in m.

Programmatic Use
Block Parameter: `lla`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

NED velocity vector [m/s] — Body velocity

`[0 0 0]` (default) | velocity vector

Body velocity with respect to Earth-centered Earth-fixed (ECEF), expressed in the north-east-down (NED) coordinate frame, specified as a vector, in m/s.

Programmatic Use
Block Parameter: `v_ned`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

CubeSat Attitude

Initial Euler angles (roll, pitch, yaw) [deg] — Initial Euler rotation angles

`[0 0 0]` (default) | vector | degrees

Initial Euler rotation angles (roll, pitch, yaw) between Body and NED coordinate frames, specified in degrees.

Programmatic Use
Block Parameter: `euler`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

Initial body angular rates [deg/s] — Initial angular rates

`[0 0 -0.05168]` (default) | vector

Initial angular rates with respect to NED frame, expressed in Body frame, specified as a vector.

Programmatic Use
Block Parameter: `pqr`
Type: character vector
Values: scalar
Default: `'[0 0 0]'`

Pointing mode — CubeSat vehicle pointing mode

`Earth (Nadir) Pointing` (default) | `Sun Tracking` | `Custom Pointing` | `Standby (Off)`

CubeSat vehicle pointing mode, specified as `Earth (Nadir) Pointing`, `Sun Tracking`, or `Custom Pointing`. The CubeSat vehicle uses the pointing mode for precise attitude control. For no attitude control, select `Standby (Off)`.

Programmatic Use
Block Parameter: `pointingMode`
Type: character vector
Values: `'Earth (Nadir) Pointing'` | `'Sun Tracking'` | `'Custom Pointing'` | `'Standby (Off)'`
Default: `'Earth (Nadir) Pointing'`

Primary alignment vector (Body wrt B_{CM}) — Primary alignment vector
`Dialog` (default) | `Input port`

Primary alignment vector, in Body frame, to align with primary constraint vector.

Dependencies

- Selecting `Dialog` enables a text box in which you specify the primary alignment vector. The default value is `[0 0 1]`.
- Selecting `Input port` enables the 1st $Alignment_{Body}$ input port, at which you specify the primary alignment vector.

Programmatic Use
Block Parameter: `firstAlign`
Type: character vector
Values: vector
Default: `'[0 0 1]'`

Programmatic Use
Block Parameter: `firstAlignExt`
Type: character vector
Values: `'Input port'` | `'Dialog'`
Default: `'Dialog'`

Secondary alignment vector (Body wrt B_{CM}) — Secondary alignment vector
`Dialog` (default) | `Input port`

Secondary alignment vector, in Body frame, to align with secondary constraint vector.

Dependencies

- Selecting `Dialog` enables a text box in which you specify the secondary alignment vector. The default value is `[0 1 0]`.
- Selecting `Input port` enables the 2nd $Alignment_{Body}$ input port, at which you specify the secondary alignment vector.

Programmatic Use
Block Parameter: `secondAlign`
Type: character vector
Values: vector
Default: `'[0 1 0]'`

Programmatic Use
Block Parameter: `secondAlignExt`
Type: character vector
Values: `'Input port'` | `'Dialog'`
Default: `'Dialog'`

Constraint coordinate system — Constraint coordinate system

`ECI Axes` (default) | `ECEF Axes` | `NED Axes` | `Body-Fixed Axes`

Constraint coordinate system, specified as `ECI Axes`, `ECEF Axes`, `NED Axes`, or `Body-Fixed Axes`.

Programmatic Use
Block Parameter: `constraintCoord`
Type: character vector
Values: `'ECI Axes'` | `'ECEF Axes'` | `'NED Axes'` | `'Body-Fixed Axes'`
Default: `'ECI Axes'`

Primary constraint vector (wrt B_{CM}) — Primary constraint vector
`Dialog` (default) | `Input port`

Primary constraint vector, in the Body frame, to align with the primary alignment vector.

Dependencies

- This parameter is disabled when **Pointing mode** is `Earth (Nadir) Pointing` or `Sun Tracking`.
- Selecting `Dialog` enables a text box in which you specify the primary constraint vector. The default value is `[1 0 0]`.
- Selecting `Input port` enables the 1st $\text{constraint}_{\text{Body}}$ input port, at which you specify the primary constraint vector.

Programmatic Use
Block Parameter: `firstRef`
Type: character vector
Values: vector
Default: `'[1 0 0]'`

Programmatic Use
Block Parameter: `firstRefExt`
Type: character vector
Values: `'Input port'` | `'Dialog'`
Default: `'Dialog'`

Secondary constraint vector (wrt B_{CM}) — Secondary constraint vector
`Dialog` (default) | `Input port`

Secondary constraint vector, in the Body frame, to align with the secondary alignment vector.

Dependencies

- Selecting `Dialog` enables a text box in which you specify the secondary constraint vector. The default value is `[0 0 1]`.
- Selecting `Input port` enables the 2nd $\text{constraint}_{\text{Body}}$ input port, at which you specify the secondary constraint vector.

Programmatic Use
Block Parameter: `secondRef`
Type: character vector
Values: vector
Default: `'[0 0 1]'`

Programmatic Use
Block Parameter: `secondRefExt`
Type: character vector
Values: `'Input port'` | `'Dialog'`
Default: `'Dialog'`

Mission Analysis

Analysis run time source — Source of run time for mission analysis live script

`Dialog` (default) | `Model Stop Time`

Source of run time for mission analysis live script, specified as:

- `Dialog` — Defined in **Run time** parameter.
- `Model Stop Time` — Defined in model configuration parameter **Stop Time**.

Programmatic Use
Block Parameter: `missionRTSource`
Type: character vector
Values: `'Dialog'` | `'Model StopTime'`
Default: `'Dialog'`

Run time [sec] — Run time for mission analysis live script

`6*60*60` (default) | scalar

Run time for mission analysis live script, specified as a scalar.

Programmatic Use
Block Parameter: `missionRT`
Type: character vector
Values: scalar
Default: `'6*60*60'`

Ground station geodetic latitude, longitude [deg, deg] — Ground station location

[42, -71] (default) | vector

Ground station location, specified as a vector, in geodetic latitude and longitude in deg, deg.

Programmatic Use
Block Parameter: `missionGS`
Type: character vector
Values: vector
Default: `'[42, -71]'`

Run TOI analysis — Enable time of interest mission analysis

on (default) | off

Select this check box to enable time of interest analysis in mission analysis.live script

Programmatic Use
Block Parameter: `missionTOICheck`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Time of interest [Julian date] — Time of interest for mission analysis live script

[ ] (default) | Julian date

Time of interest mission analysis, specified as a Julian date. To use the simulation start date, enter an empty array (`[]`).

Tip To calculate the Julian date, use the `juliandate` function.

Programmatic Use
Block Parameter: `missionTOI`
Type: character vector
Values: Julian date
Default: `'[]'`

Camera field-of-view (FOV) half angle (deg) — Half angle of field of view

55 (default) | [ ] | scalar

Half angle of field of view for nadir on-pointed camera. To exclude from analysis, enter an empty array (`[]`).

Programmatic Use
Block Parameter: `missionEta`
Type: character vector
Values: `'[]'` | scalar
Default: `'55'`

Live script file name — File name for mission analysis live script report

blank entry (default) | live script file name

File name for mission analysis live script report, generated as a live script. To create a default mission analysis report with the format `CubeSatMissionReport_`*`currentdate`*`.mlx`, leave the parameter blank. To create a live script of the mission analysis report, click the **Create Live Script Report** button.

Dependencies

To create the live script with the specified file name, click the **Create Live Script Report** button. If this parameter is blank, the block creates a live script with a default file name.

Programmatic Use
Block Parameter: `missionName`
Type: character vector
Values: blank entry | file name
Default: blank entry

Create Live Script Report — Analyze mission and create live script report

button

To analyze mission and create report in live script format, click this button. To create a default mission analysis report with the format `CubeSatMissionReport_`*`currentdate`*`.mlx`, leave the parameter blank. To create a live script of the mission analysis report, click the **Create Live Script Report** button.

Dependencies

To create the live script with the file name specified in **Live script file name**, click the **Create Live Script Report** button. If **Live script file name** is blank, the block creates a live script with a default file name.

Version History

Introduced in R2019a

R2021a: CubeSat Vehicle now propagates in the ECI coordinate frame

Behavior changed in R2021a

The CubeSat Vehicle now propagates in the ECI coordinate frame using Earth orientation parameters data from the `aeroiersdata.mat` file. Results differ from previous releases, but are more accurate than with previous versions of the block.

References

[1] Wertz, James R, David F. Everett, and Jeffery J. Puschell. Space Mission Engineering: The New Smad. Hawthorne, CA: Microcosm Press, 2011. Print.

See Also

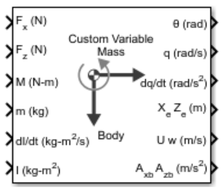
Attitude Profile | Orbit Propagator | `ecef2eci` | `eci2ecef` | `ijk2keplerian` | `juliandate` | `keplerian2ijk` | `siderealTime`

Topics

"Model and Simulate CubeSats" on page 2-73

Custom Variable Mass 3DOF (Body Axes)

Implement three-degrees-of-freedom equations of motion of custom variable mass with respect to body axes



Libraries:
Aerospace Blockset / Equations of Motion / 3DOF

Description

The Custom Variable Mass 3DOF (Body Axes) block implements three-degrees-of-freedom equations of motion of custom variable mass with respect to body axes. It considers the rotation in the vertical plane of a body-fixed coordinate frame about a flat Earth reference frame. For more information about the rotation and equations of motion, see “Algorithms” on page 5-249.

Ports

Input

F_x — Applied force along *x*-axis
scalar

Applied force along the body *x*-axis, specified as a scalar, in the units selected in **Units**.

Data Types: `double`

F_z — Applied force along *z*-axis
scalar

Applied force along the body *z*-axis, specified as a scalar.

Data Types: `double`

M — Applied pitching moment
scalar

Applied pitching moment, specified as a scalar.

Data Types: `double`

dm/dt — Rate of change of mass
scalar

Rate of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Data Types: `double`

dI/dt — Rate of change of inertia tensor
scalar

Rate of change of inertia tensor, I_{yy}, specified as scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

I — Inertia tensor
scalar

Inertia tensor, specified as a scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

g — Gravity
scalar

Gravity, specified as a scalar.

Dependencies

To enable this port, set **Gravity source** to `External`.

Data Types: `double`

V_{re} — Relative velocity
two-element vector

Relative velocity at which mass is accreted to or ablated from the body in body-fixed axes, specified as a two-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

θ — Pitch attitude
scalar

Pitch attitude, within ±pi, returned as a scalar, in radians (θ).

Data Types: `double`

q — Pitch angular rate
scalar

Pitch angular rate, returned as a scalar, in radians per second.

Data Types: `double`

dq/dt — Pitch angular acceleration
scalar

Pitch angular acceleration, returned as a scalar, in radians per second squared.

Data Types: `double`

X_eZ_e — Location of body
two-element vector

Location of the body in the flat Earth reference frame, (*Xe, Ze*), returned as a two-element vector.

Data Types: `double`

U w — Velocity of body
two-element vector

Velocity of the body resolved into the body-fixed coordinate frame, (*u, w*), returned as a two-element vector.

Data Types: `double`

$A_{xb}A_{zb}$ — Acceleration of body
two-element vector

Acceleration of the body with respect to the body-fixed coordinate frame, (*Ax, Az*), returned as a two-element vector.

Data Types: `double`

$A_{xe}A_{ze}$ — Acceleration of body
two-element vector

Accelerations of the body with respect to the inertial (flat Earth) coordinate frame, returned as a two-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this port, select the **Include inertial acceleration** check box.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
Metric (MKS)	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
English (Velocity in ft/s)	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
English (Velocity in kts)	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Axes — Body or wind axes

`Body` (default) | `Wind`

Body or wind axes, specified as `Wind` or `Body`

Programmatic Use
Block Parameter: `axes`
Type: character vector
Values: `Wind` | `Body`
Default: `Body`

Mass type — Mass type
`Custom Variable` (default) | `Fixed` | `Simple Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 3DOF (Body Axes) • 3DOF (Wind Axes)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 3DOF (Body Axes) • Simple Variable Mass 3DOF (Wind Axes)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 3DOF (Body Axes) • Custom Variable Mass 3DOF (Wind Axes)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`

Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Initial velocity — Initial velocity of body

`100` (default) | scalar

Initial velocity of the body, (V_0), specified as a scalar.

Programmatic Use
Block Parameter: `v_ini`
Type: character vector
Values: `'100'` | scalar
Default: `'100'`

Initial body attitude — Initial pitch altitude

`0` (default) | scalar

Initial pitch attitude of the body, (θ_0), specified as a scalar.

Programmatic Use
Block Parameter: `theta_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial body rotation rate — Initial pitch rotation rate

`0` (default) | scalar

Initial pitch rotation rate, (q_0), specified as a scalar.

Programmatic Use
Block Parameter: `q_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial incidence — Initial angle

`0` (default) | scalar

Initial angle between the velocity vector and the body, (α_0), specified as a scalar.

Programmatic Use
Block Parameter: `alpha_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial position (x,z) — Initial location

`[0 0]` (default) | two-element vector

Initial location of the body in the flat Earth reference frame, specified as a two-element vector.

Programmatic Use
Block Parameter: `pos_ini`
Type: character vector
Values: `'[0 0]'` | two-element vector
Default: `'[0 0]'`

Gravity Source — Gravity source

`Internal` (default) | `External`

Gravity source, specified as:

`External`	Variable gravity input to block
`Internal`	Constant gravity specified in mask

Programmatic Use
Block Parameter: `g_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Acceleration due to gravity — Gravity source

`9.81` (default) | scalar

Acceleration due to gravity, specified as a double scalar and used if internal gravity source is selected. If gravity is to be neglected in the simulation, this value can be set to `0`.

Dependencies

- To enable this parameter, set **Gravity Source** to `Internal`.

Programmatic Use
Block Parameter: `g`
Type: character vector
Values: `'9.81'` | scalar
Default: `'9.81'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `'off'`

Include inertial acceleration — Include inertial acceleration port

off (default) | on

Select this check box to add an inertial acceleration in flat Earth frame output port. You typically connect this signal to the accelerometer.

Dependencies

To enable the $\mathbf{A_{xe}A_{ze}}$ port, select this parameter.

Programmatic Use
Block Parameter: abi_flag
Type: character vector
Values: 'off' | 'on'
Default: 'off'

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- The number of names must match the number of states, as shown for each item, or be empty. Set all or none of the block states.
- To assign names to single-variable states, enter unique names between quotes, for example, 'q' or "q".
- To assign names to two-variable states, enter a comma-separated list surrounded by braces, for example, {'Xe','Ze'}.
- If a state parameter is empty (' '), no name is assigned.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array of character vectors, or string.

Velocity: e.g., {'u, 'w'} — Velocity state name

'' (default) | comma-separated list surrounded by braces

Velocity state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: vel_statename
Type: character vector
Values: '' | comma-separated list surrounded by braces
Default: ''

Position: e.g., {'Xe', 'Ze'} — Position state name

'' (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: pos_statename
Type: character vector
Values: '' | comma-separated list surrounded by braces
Default: ''

Pitch angular rate e.g., 'q' — Pitch angular rate state name

`''` (default)

Pitch angular rate state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `q_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Pitch attitude: e.g., 'theta' — Pitch attitude state name

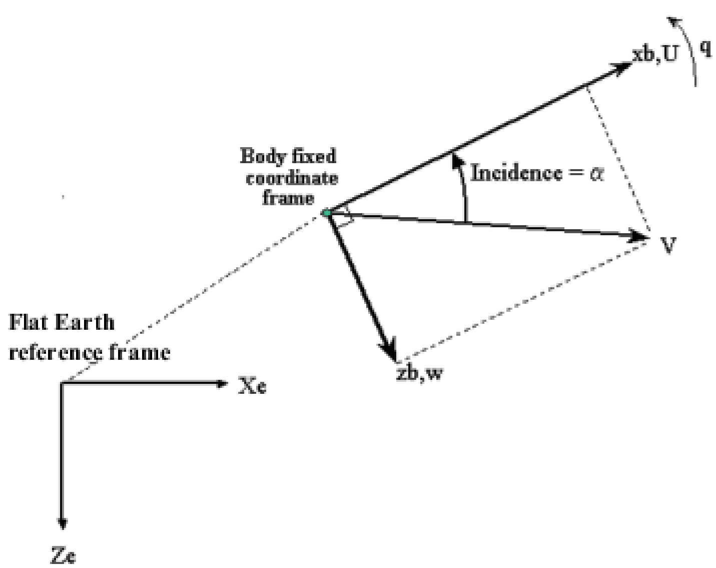
`''` (default)

Pitch attitude state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `theta_statename`
Type: character vector | string
Values: `''`
Default: `''`

Algorithms

The Custom Variable Mass 3DOF (Body Axes) block considers the rotation in the vertical plane of a body-fixed coordinate frame about a flat Earth reference frame.



The equations of motion are

$$A_{xb} = \dot{u} = A_{xe} - qw$$
$$A_{zb} = \dot{w} = A_{ze} + qu$$
$$A_{xe} = \frac{(F_x - \dot{m}u_{re})}{m} - g\sin\theta$$
$$A_{ze} = \frac{(F_z - \dot{m}w_{re})}{m} + g\cos\theta$$
$$\dot{X}_e = u\cos\theta + w\sin\theta$$
$$\dot{Z}_e = -u\sin\theta + w\cos\theta$$
$$\dot{q} = \frac{M_y - \dot{I}_{yy}q}{I_{yy}}$$
$$\dot{\theta} = q$$

where the applied forces are assumed to act at the center of gravity of the body. Input variables are F_x, F_z, M_y, $\dot{m}$ (dm/dt), m, $\dot{I}$ (dI_{yy}/dt), and I_{yy}. u_{re}, w_{re}, and g are optional input variables.

Version History

Introduced in R2006a

R2021b: Custom Variable Mass 3DOF (Body Axes) Block Changes

Behavior changed in R2021b

The 3DOF equations of motion have been updated. Existing models created prior to R2021b that contain 3DOF equations of motion blocks continue to run. If you replace a pre-R2021b version of a 3DOF equation of motion block with an R2021b or later version, your updated model might have a higher tendency for algebraic loops. For an example of how to remove algebraic loops using unit delays, see “Remove Algebraic Loops”. For further information about algebraic loops, see “Identify Algebraic Loops in Your Model”.

Extended Capabilities

C/C++ Code Generation

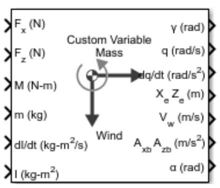
Generate C and C++ code using Simulink® Coder™.

See Also

3DOF (Body Axes) | Incidence & Airspeed | Simple Variable Mass 3DOF (Body Axes)

Custom Variable Mass 3DOF (Wind Axes)

Implement three-degrees-of-freedom equations of motion of custom variable mass with respect to wind axes



Libraries:
Aerospace Blockset / Equations of Motion / 3DOF

Description

The Custom Variable Mass 3DOF (Wind Axes) block implements three-degrees-of-freedom equations of motion of custom variable mass with respect to wind axes. It considers the rotation in the vertical plane of a wind-fixed coordinate frame about a flat Earth reference frame. For more information about the rotation and equations of motion, see “Algorithms” on page 5-258.

Ports

Input

F$_x$ — Applied force along wind *x*-axis
scalar

Applied force along the wind *x*-axis, specified as a scalar, in the units selected in **Units**.

Data Types: `double`

F$_z$ — Applied force along wind *z*-axis
scalar

Applied force along the wind *z*-axis, specified as a scalar.

Data Types: `double`

M — Applied pitching moment
scalar

Applied pitching moment, specified as a scalar.

Data Types: `double`

dm/dt — Rate of change of mass
scalar

Rate of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Data Types: `double`

dI/dt — Rate of change of inertia tensor
scalar

Rate of change of inertia tensor, I_{yy}, specified as scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

I — Inertia tensor
scalar

Inertia tensor, specified as a scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

g — Gravity
scalar

Gravity, specified as a scalar.

Dependencies

To enable this port, set **Gravity source** to `External`.

Data Types: `double`

V_{re} — Relative velocity
two-element vector

Relative velocity at which mass is accreted to or ablated from the body in body-fixed axes, specified as a two-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

γ — Flight path angle
scalar

Flight path angle, within ±pi, returned as a scalar, in radians.

Data Types: `double`

q — Pitch angular rate
scalar

Pitch angular rate, returned as a scalar, in radians per second.

Data Types: `double`

dq/dt — Pitch angular acceleration
scalar

Pitch angular acceleration, returned as a scalar, in radians per second squared.

Data Types: `double`

X_eZ_e — Location of body
two-element vector

Location of the body in the flat Earth reference frame, (*Xe, Ze*), returned as a two-element vector.

Data Types: `double`

V_w — Velocity in wind-fixed frame
two-element vector

Velocity of the body resolved into the wind-fixed coordinate frame, (*V, 0*), returned as a two-element vector.

Data Types: `double`

$A_{xb}A_{zb}$ — Acceleration of body
two-element vector

Acceleration of the body with respect to the body-fixed coordinate frame, (*Ax, Az*), returned as a two-element vector.

Data Types: `double`

α — Angle of attack
scalar

Angle of attack, returned as a scalar, in radians.

Data Types: `double`

$A_{xe}A_{ze}$ — Acceleration of body
two-element vector

Accelerations of the body with respect to the inertial (flat Earth) coordinate frame, returned as a two-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this port, select the **Include inertial acceleration** check box.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Axes — Body or wind axes

`Wind` (default) | `Body`

Body or wind axes, specified as `Wind` or `Body`

Programmatic Use
Block Parameter: `axes`
Type: character vector
Values: `Wind` | `Body`
Default: `Wind`

Mass type — Mass type

`Custom Variable` (default) | `Simple Variable` | `Fixed`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 3DOF (Body Axes) • 3DOF (Wind Axes)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 3DOF (Body Axes) • Simple Variable Mass 3DOF (Wind Axes)

Mass Type	Description	Default for
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 3DOF (Body Axes) • Custom Variable Mass 3DOF (Wind Axes)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Initial airspeed — Initial speed

`100` (default) | scalar

Initial speed of the body, (V_0), specified as a scalar.

Programmatic Use
Block Parameter: `V_ini`
Type: character vector
Values: `'100'` | scalar
Default: `'100'`

Initial flight path angle — Initial flight path angle

`0` (default) | scalar

Initial flight path angle of the body, (γ_0), specified as a scalar.

Programmatic Use
Block Parameter: `gamma_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial body rotation rate — Initial pitch rotation rate

`0` (default) | scalar

Initial pitch rotation rate, (q_0), specified as a scalar.

Programmatic Use
Block Parameter: `q_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial incidence — Initial angle

`0` (default) | scalar

Initial angle between the velocity vector and the body, (α_0), specified as a scalar.

Programmatic Use
Block Parameter: `alpha_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial position (x,z) — Initial location

`[0 0]` (default) | two-element vector

Initial location of the body in the flat Earth reference frame, specified as a two-element vector.

Programmatic Use
Block Parameter: `pos_ini`
Type: character vector
Values: `'[0 0]'` | two-element vector
Default: `'[0 0]'`

Gravity Source — Gravity source

`Internal` (default) | `External`

Gravity source, specified as:

`External`	Variable gravity input to block
`Internal`	Constant gravity specified in mask

Programmatic Use
Block Parameter: `g_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Acceleration due to gravity — Gravity source

`9.81` (default) | scalar

Acceleration due to gravity, specified as a double scalar and used if internal gravity source is selected. If gravity is to be neglected in the simulation, this value can be set to `0`.

Dependencies

- To enable this parameter, set **Gravity Source** to `Internal`.

Programmatic Use
Block Parameter: `g`
Type: character vector
Values: `'9.81'` | scalar
Default: `'9.81'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `'off'`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration in flat Earth frame output port. You typically connect this signal to the accelerometer.

Dependencies

To enable the $\mathbf{A_{xe}A_{ze}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `V_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Position: e.g., {'Xe', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pos_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rate: e.g., 'q' — Body rotation state name

`''` (default) | scalar

Body rotation rate state names, specified as a character vector or string.

Programmatic Use
Block Parameter: `q_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Flight path angle: e.g., 'gamma' — Flight path angle state name

`''` (default)

Flight path angle state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `gamma_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

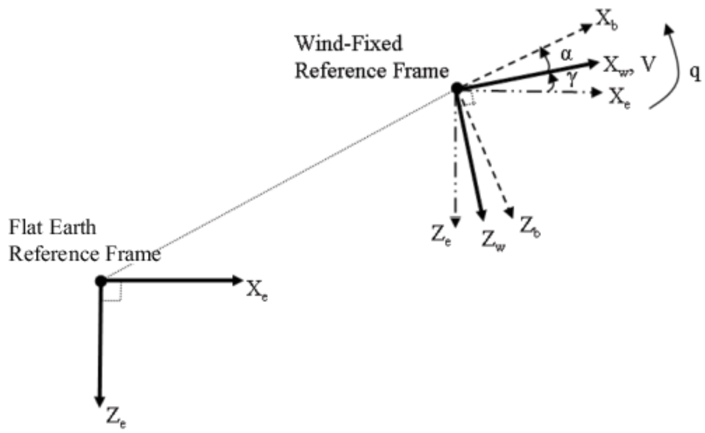
`''` (default) | scalar

Incidence angle state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Algorithms

The block considers the rotation in the vertical plane of a wind-fixed coordinate frame about a flat Earth reference frame.



The equations of motion are

$$A_{xb} = A_{xe} - qV\sin\alpha$$

$$A_{zb} = A_{ze} + qV\cos\alpha$$

$$A_{xe} = \left(\frac{F_x}{m} - g\sin\gamma\right)\cos\alpha - \left(\frac{F_z}{m} + g\cos\gamma\right)\sin\alpha$$

$$A_{ze} = \left(\frac{F_x}{m} - g\sin\gamma\right)\sin\alpha + \left(\frac{F_z}{m} + g\cos\gamma\right)\cos\alpha$$

$$\dot{V} = \frac{(F_x + \dot{m}u_{re})}{m} - g\sin\gamma$$

$$\dot{X}_e = V\cos\gamma$$

$$\dot{Z}_e = -V\sin\gamma$$

$$\dot{q} = \frac{M_y - \dot{I}_{yy}q}{I_{yy}}$$

$$\dot{\gamma} = q - \dot{\alpha}$$

$$\dot{\alpha} = \frac{(F_z + \dot{m}w_{re})}{mV} + \frac{g}{V}\cos\gamma + q$$

where the applied forces are assumed to act at the center of gravity of the body. Input variables are wind-axes forces F_x and F_z, body moment M_y, $\dot{m}$ (dm/dt), m, $\dot{I}$ (dI_{yy}/dt), and I_{yy}. u_{re}, w_{re}, and g are optional input variables.

Version History

Introduced in R2006a

R2021b: Custom Variable Mass 3DOF (Wind Axes) Block Changes

Behavior changed in R2021b

The 3DOF equations of motion have been updated. Existing models created prior to R2021b that contain 3DOF equations of motion blocks continue to run. If you replace a pre-R2021b version of a 3DOF equation of motion block with an R2021b or later version, your updated model might have a higher tendency for algebraic loops. For an example of how to remove algebraic loops using unit delays, see “Remove Algebraic Loops”. For further information about algebraic loops, see “Identify Algebraic Loops in Your Model”.

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. New York: John Wiley & Sons, 1992.

Extended Capabilities

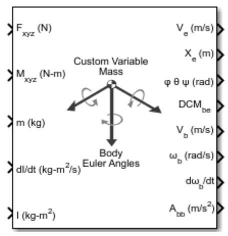
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

3DOF (Body Axes) | 3DOF (Wind Axes) | 4th Order Point Mass (Longitudinal) | Custom Variable Mass 3DOF (Body Axes) | Simple Variable Mass 3DOF (Body Axes) | Simple Variable Mass 3DOF (Wind Axes)

Custom Variable Mass 6DOF (Euler Angles)

Implement Euler angle representation of six-degrees-of-freedom equations of motion of custom variable mass



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Custom Variable Mass 6DOF (Euler Angles) block implements the Euler angle representation of six-degrees-of-freedom equations of motion of custom variable mass. It considers the rotation of a body-fixed coordinate frame (X_b, Y_b, Z_b) about a flat Earth reference frame (X_e, Y_e, Z_e). For more information on Euler angles, see "Algorithms" on page 5-268.

Limitations

The block assumes that the applied forces act at the center of gravity of the body.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt — Rates of change of mass
three-element vector

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 matrix

Rate of change of inertia tensor matrix, specified as a 3-by-3 matrix.

Data Types: `double`

I — Inertia tensor matrix
3-by-3 matrix

Inertia tensor matrix, specified as a 3-by-3 matrix.

Data Types: `double`

V_{re} — Relative velocities
three-element vector

One or more relative velocities at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

φ θ ψ (rad) — Euler rotation angles
three-element vector

Euler rotation angles [roll, pitch, yaw], returned as a three-element vector, in radians.

Data Types: `double`

DCM_{be} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity in body-fixed frame
three-element vector

Velocity in body-fixed frame, returned as a three-element vector.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
Metric (MKS)	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
English (Velocity in ft/s)	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
English (Velocity in kts)	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass type — Mass type

`Custom Variable` (default) | `Simple Variable` | `Fixed`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)

Mass Type	Description	Default for
Custom Variable	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Representation — Equations of motion representation

`Euler Angles` (default) | `Quaternion`

Equations of motion representation, specified according to the following table.

Representation	Description
Euler Angles	Use Euler angles within equations of motion.
Quaternion	Use quaternions within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-268.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Euler Angles` | `Quaternion`
Default: `'Euler Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial velocity in body axes [U,v,w] — Velocity in body axes

[0 0 0] (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

[0 0 0] (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

[0 0 0] (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Include mass flow relative velocity — Mass flow relative velocity port

off (default) | on

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

off (default) | on

Select this check box to add an inertial acceleration port.

Dependencies

To enable the **$A_{b\ ff}$** port, select this parameter.

Programmatic Use

Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use

Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use

Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Euler rotation angles: e.g., {'phi', 'theta', 'psi'} — Euler rotation state name

'' (default) | comma-separated list surrounded by braces

Euler rotation angle state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `eul_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

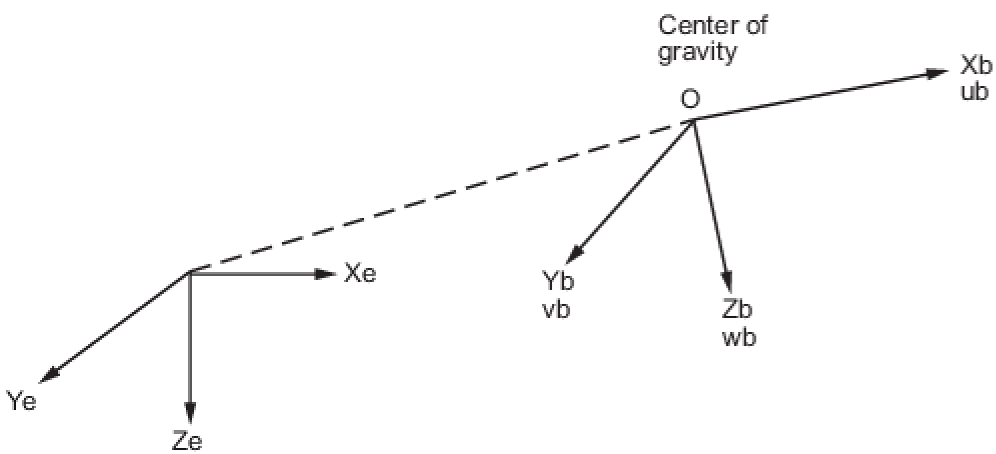
'' (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Algorithms

The origin of the body-fixed coordinate frame is the center of gravity of the body. The body is assumed to be rigid, which eliminates the need to consider the forces acting between individual elements of mass. The flat Earth reference frame is considered inertial, an excellent approximation that allows the forces due to the Earth's motion relative to the “fixed stars” to be neglected.



Flat Earth reference frame

The translational motion of the body-fixed coordinate frame is given below, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the body-fixed frame. Vre_b is the relative velocity in the body axes at which the mass flow ($\dot{m}$) is ejected or added to the body-fixed axes.

$$\bar{F}_b = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m(\dot{\bar{V}}_b + \bar{\omega} \times \bar{V}_b) + \dot{m}\bar{V}re_b$$

$$A_{be} = \frac{\bar{F}_b - \dot{m}\bar{V}_{re_b}}{m}$$

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{w}_b \end{bmatrix} = \frac{\bar{F}_b - \dot{m}\bar{V}_{re_b}}{m} - \bar{\omega} \times \bar{V}_b$$

$$\bar{V}_b = \begin{bmatrix} u_b \\ v_b \\ w_b \end{bmatrix}, \bar{\omega} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

The rotational dynamics of the body-fixed frame are given below, where the applied moments are $[L\ M\ N]^T$, and the inertia tensor I is with respect to the origin O.

$$\bar{M}_B = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\bar{\omega}} + \bar{\omega} \times (I\bar{\omega}) + \dot{I}\,\bar{\omega}$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

$$\dot{I} = \begin{bmatrix} \dot{I}_{xx} & -\dot{I}_{xy} & -\dot{I}_{xz} \\ -\dot{I}_{yx} & \dot{I}_{yy} & -\dot{I}_{yz} \\ -\dot{I}_{zx} & -\dot{I}_{zy} & \dot{I}_{zz} \end{bmatrix}$$

The relationship between the body-fixed angular velocity vector, $[p\ q\ r]^T$, and the rate of change of the Euler angles, $[\dot{\phi}\dot{\theta}\dot{\psi}]^T$, can be determined by resolving the Euler rates into the body-fixed coordinate frame.

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} = J^{-1} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

Inverting J then gives the required relationship to determine the Euler rate vector.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = J \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & (\sin\phi\tan\theta) & (\cos\phi\tan\theta) \\ 0 & \cos\phi & -\sin\phi \\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

For more information on aerospace coordinate systems, see “About Aerospace Coordinate Systems” on page 2-8.

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

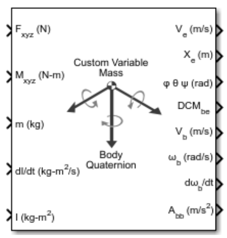
See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Topics
"About Aerospace Coordinate Systems" on page 2-8

Custom Variable Mass 6DOF (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion of custom variable mass with respect to body axes



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Custom Variable Mass 6DOF (Quaternion) block implements a quaternion representation of six-degrees-of-freedom equations of motion of custom variable mass with respect to body axes. For a description of the coordinate system and the translational dynamics, see the block description for the Custom Variable Mass 6DOF (Euler Angles) block.

Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For more information on the integration of the rate of change of the quaternion vector, see “Algorithms” on page 5-278.

Limitations

The block assumes that the applied forces act at the center of gravity of the body.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt — Rates of change of mass
three-element vector

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 matrix

Rate of change of inertia tensor matrix, specified as a 3-by-3 matrix.

Data Types: `double`

I — Inertia tensor matrix
3-by-3 matrix

Inertia tensor matrix, specified as a 3-by-3 matrix.

Data Types: `double`

V_{re} — Relative velocities
three-element vector

One or more relative velocities at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

φ θ ψ (rad) — Euler rotation angles
three-element vector

Euler rotation angles [roll, pitch, yaw], returned as a three-element vector, in radians.

Data Types: `double`

DCM_{be} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity in body-fixed frame
three-element vector

Velocity in body-fixed frame, returned as a three-element vector.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass type — Mass type

`Custom Variable` (default) | `Simple Variable` | `Fixed`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)

Mass Type	Description	Default for
Custom Variable	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Representation — Equations of motion representation

`Quaternion` (default) | `Euler Angles`

Equations of motion representation, specified according to the following table.

Representation	Description
Quaternion	Use quaternions within equations of motion.
Euler Angles	Use Euler angles within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-278.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Euler Angles` | `Quaternion`
Default: `'Euler Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial velocity in body axes [U,v,w] — Velocity in body axes

`[0 0 0]` (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

`[0 0 0]` (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Gain for quaternion normalization — Gain

`1.0` (default) | scalar

Gain to maintain the norm of the quaternion vector equal to 1.0, specified as a double scalar.

Programmatic Use
Block Parameter: `k_quat`
Type: character vector
Values: `1.0` | double scalar
Default: `1.0`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: vre_flag
Type: character vector
Values: off | on
Default: off

Include inertial acceleration — Include inertial acceleration port

off (default) | on

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: abi_flag
Type: character vector
Values: 'off' | 'on'
Default: off

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, 'velocity'.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- If a parameter is empty (' '), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

'' (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: xme_statename
Type: character vector
Values: '' | comma-separated list surrounded by braces
Default: ''

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

'' (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

'' (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

'' (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Algorithms

The integration of the rate of change of the quaternion vector is given below. The gain K drives the norm of the quaternion state vector to 1.0 should ε become nonzero. You must choose the value of this gain with care, because a large value improves the decay rate of the error in the norm, but also slows the simulation because fast dynamics are introduced. An error in the magnitude in one element of the quaternion vector is spread equally among all the elements, potentially increasing the error in the state vector.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = {}^1\!/_2 \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} + K\varepsilon \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

$$\varepsilon = 1 - (q_0^2 + q_1^2 + q_2^2 + q_3^2) .$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

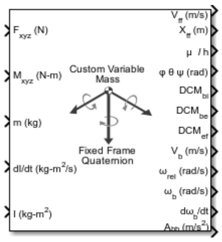
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Custom Variable Mass 6DOF ECEF (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion of custom variable mass in Earth-centered Earth-fixed (ECEF) coordinates



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Custom Variable Mass 6DOF ECEF (Quaternion) block implements a quaternion representation of six-degrees-of-freedom equations of motion of custom variable mass in Earth-centered Earth-fixed (ECEF) coordinates. It considers the rotation of a Earth-centered Earth-fixed (ECEF) coordinate frame (X_{ECEF}, Y_{ECEF}, Z_{ECEF}) about an Earth-centered inertial (ECI) reference frame (X_{ECI}, Y_{ECI}, Z_{ECI}). The origin of the ECEF coordinate frame is the center of the Earth. For more information on the ECEF coordinate frame, see “Algorithms” on page 5-292.

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Limitations

- This implementation assumes that the applied forces act at the center of gravity of the body.
- This implementation generates a geodetic latitude that lies between ±90 degrees, and longitude that lies between ±180 degrees. Additionally, the MSL altitude is approximate.
- The Earth is assumed to be ellipsoidal. By setting flattening to 0.0, a spherical planet can be achieved. The Earth's precession, nutation, and polar motion are neglected. The celestial longitude of Greenwich is Greenwich Mean Sidereal Time (GMST) and provides a rough approximation to the sidereal time.
- The implementation of the ECEF coordinate system assumes that the origin is at the center of the planet, the *x*-axis intersects the Greenwich meridian and the equator, the *z*-axis is the mean spin axis of the planet, positive to the north, and the *y*-axis completes the right-handed system.
- The implementation of the ECI coordinate system assumes that the origin is at the center of the planet, the *x*-axis is the continuation of the line from the center of the Earth toward the vernal equinox, the *z*-axis points in the direction of the mean equatorial plane's north pole, positive to the north, and the *y*-axis completes the right-handed system.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector, in body axes.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector, in body axes.

Data Types: `double`

dm/dt — Rates of change of mass
three-element vector

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a three-element vector.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 matrix

Rate of change of inertia tensor matrix, specified as a 3-by-3 matrix.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

I — Inertia tensor matrix
3-by-3 matrix

Inertia tensor matrix, specified as a 3-by-3 matrix.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

$L_G(0)$ — Initial celestial longitude of Greenwich
scalar

Greenwich meridian initial celestial longitude angle, specified as a scalar.

Dependencies

To enable this port

- Set **Celestial longitude of Greenwich** to `External`.
- set **Planet model** to `Earth`.

Data Types: `double`

L_{PM}(0) — Prime meridian initial celestial longitude angle
scalar

Prime meridian initial celestial longitude angle, specified as a scalar.

Dependencies

To enable this port

- Set **Celestial longitude of prime meridian** to `External`.
- Set **Planet model** to `Custom`.

Data Types: `double`

V_{re} — Relative velocities
three-element vector

One or more relative velocities at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_{ff} — Velocity of body with respect to ECEF frame,
three-element vector

Velocity of body with respect to ECEF frame, expressed in ECEF frame, returned as a three-element vector.

Data Types: `double`

X_{ff} — Position in ECEF reference frame
three-element vector

Position in ECEF reference frame, returned as a three-element vector.

Data Types: `double`

μ l h — Position in geodetic latitude, longitude, and altitude
three-element vector | M-by-3 array

Position in geodetic latitude, longitude, and altitude, in degrees, returned as a three-element vector or M-by-3 array, in selected units of length, respectively.

Data Types: `double`

φ θ Ψ (rad) — Body rotation angles
three-element vector

Body rotation angles [roll, pitch, yaw], returned as a three-element vector, in radians. Euler rotation angles are those between body and NED coordinate systems.

Data Types: `double`

DCM_{bi} — Coordinate transformation from ECI axes
3-by-3 matrix

Coordinate transformation from ECI axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

DCM_{be} — Coordinate transformation from NED axes to body-fixed axes
3-by-3 matrix

Coordinate transformation from NED axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

DCM_{ef} — Coordinate transformation from fixed-frame axes
3-by-3 matrix

Coordinate transformation from fixed-frame axes to NED axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity of body with respect to fixed-frame
three-element vector

Velocity of body with respect to fixed-frame, returned as a three-element vector.

Data Types: `double`

ω_{rel} — Relative angular rates of body with respect to NED frame
three-element vector

Relative angular rates of body with respect to NED frame, expressed in body frame and returned as a three-element vector, in radians per second.

Data Types: `double`

ω_b — Angular rates of body with respect to inertial frame
three-element vector

Angular rates of the body with respect to inertial frame, expressed in body frame and returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations of the body with respect to inertial frame
three-element vector

Angular accelerations of the body with respect to inertial frame, expressed in body frame and returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations of the body with respect to the fixed-frame, returned as a three-element vector.

Data Types: `double`

$A_{b\ ff}$ — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to fixed-frame, returned as a three-element vector.

Dependencies

To enable this point, **Include inertial acceleration**.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass type — Mass type

`Custom Variable` (default) | `Fixed` | `Simple Variable`

Select the type of mass to use:

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Initial position in geodetic latitude, longitude and altitude [mu,l,h] — Initial location of rigid body

`[0 0 0]` (default) | three-element vector

Initial location of the rigid body in the geodetic reference frame, specified as a three-element vector. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Programmatic Use
Block Parameter: `xg_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector

Default: `'[0 0 0]'`

Initial velocity in body axes [U,v,w] — Velocity in body axes

`[0 0 0]` (default) | three-element vector

Initial velocity of the body with respect to the ECEF frame, expressed in the body frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

`[0 0 0]` (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

Planet

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Equatorial radius of planet — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator, specified as a double scalar, in the same units as the desired units for the ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `'6378137'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `F`

Type: character vector
Values: double scalar
Default: '1/298.257223563'

Rotational rate — Rotational rate

7292115e-11 (default) | scalar

Rotational rate of the planet, specified as a scalar, in rad/s.

Dependencies

To enable this parameter, set **Planet model** to Custom.

Programmatic Use
Block Parameter: w_E
Type: character vector
Values: double scalar
Default: '7292115e-11'

Celestial longitude of Greenwich source — Source of Greenwich meridian initial celestial longitude

Internal (default) | External

Source of Greenwich meridian initial celestial longitude, specified as:

Internal	Use celestial longitude value from **Celestial longitude of Greenwich**.
External	Use external input for celestial longitude value.

Dependencies

- To enable this parameter, set **Planet model** to Earth.
- Setting this parameter to External enables the $\mathbf{L_G(0)}$ port.
- If **Planet model** is set to Custom, the parameter name changes to **Celestial longitude of prime meridian source**.

Programmatic Use
Block Parameter: angle_in
Type: character vector
Values: 'Internal' | 'External'
Default: 'Internal'

Celestial longitude of Greenwich [deg] — Initial angle

0 (default) | scalar

Initial angle between Greenwich meridian and the *x*-axis of the inertial frame, specified as a double scalar.

Dependencies

- To enable this parameter, set:
  - **Celestial longitude of Greenwich source** to Internal.

  - **Planet model** to `Earth (WGS84)`.
- If **Planet model** is set to `Custom`, the parameter name changes to **Celestial longitude of prime meridian [deg]**.

Programmatic Use

Block Parameter: `LPM0`
Type: character vector
Values: double scalar
Default: `'0'`

Celestial longitude of prime meridian source — Source of prime meridian initial celestial longitude
`Internal` (default) | `External`

Source of prime meridian initial celestial longitude, specified as:

`Internal`	Use celestial longitude value from **Celestial longitude of prime meridian**.
`External`	Use external input for celestial longitude value.

Dependencies

- To enable this parameter, set **Planet model** to `Custom`.
- Setting this parameter to `External` enables the $\mathbf{L_{PM}(0)}$ port.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich source**.

Programmatic Use

Block Parameter: `angle_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Celestial longitude of prime meridian [deg] — Initial angle
`0` (default) | scalar

Initial angle between prime meridian and the *x*-axis of the ECI frame, specified as a double scalar.

Dependencies

- To enable this parameter, set:
  - **Celestial longitude of prime meridian source** to `Internal`.
  - **Planet model** to `Custom`.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich [deg]**.

Programmatic Use

Block Parameter: `LPM0`
Type: character vector
Values: double scalar
Default: `'0'`

State Attributes

Assign a unique name to each state. Use state names instead of block paths throughout the linearization process.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces

Default: `''`

Fixed frame position: e.g., {'Xff', 'Yff', 'Zff'} — ECEF position state name

`''` (default) | comma-separated list surrounded by braces

ECEF position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `posfixedframe_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Inertial position: e.g., {'Xinertial', 'Yinertial', 'Zinertial'} — Inertial position state names

`''` (default) | comma-separated list surrounded by braces

Inertial position state names, specified as a comma-separated list surrounded by braces.

Default value is `''`.

Programmatic Use
Block Parameter: `posinertial_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Celestial longitude of Greenwich: e.g., 'LG' — Celestial longitude state name

`''` (default) | character vector

Celestial longitude of Greenwich state name, specified as a character vector.

Dependencies

- To enable this parameter, set:**Planet model** to `Earth (WGS84)`.
- If **Planet model** is set to `Custom`, the parameter name changes to **Celestial longitude of prime meridian: e.g., 'LPM'**.

Programmatic Use
Block Parameter: `LPM_statename`
Type: character vector
Values: `''` | scalar
Default: `''`

Celestial longitude of prime meridian: e.g., 'LPM' — Celestial longitude prime meridian
`''` (default) | character vector

Celestial longitude of prime meridian state name, specified as a character vector.

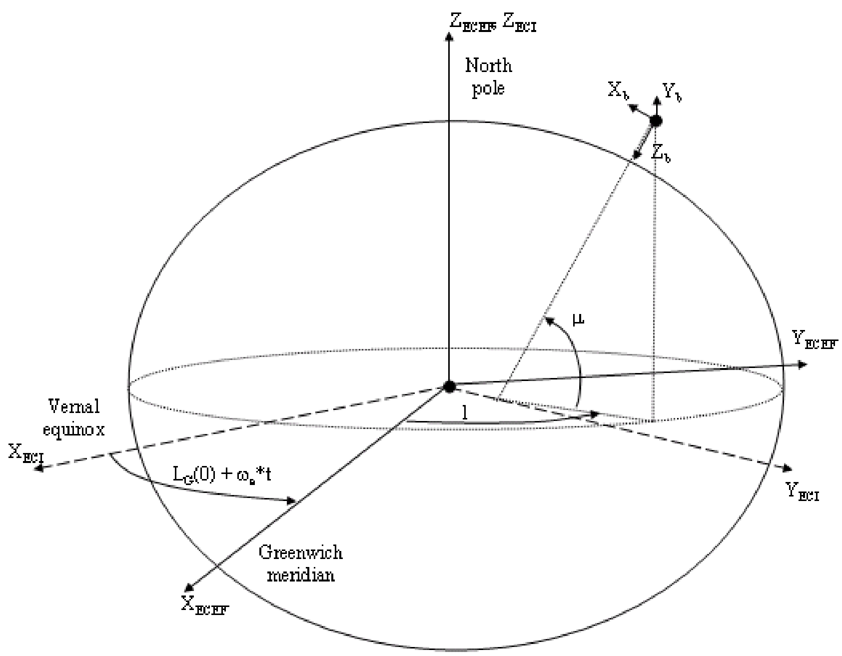
Dependencies

- To enable this parameter, set:**Planet model** to `Custom`.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich: e.g., 'LG'**.

Programmatic Use
Block Parameter: `LPM_statename`
Type: character vector
Values: `''` | scalar
Default: `''`

Algorithms

The origin of the ECEF coordinate frame is the center of the Earth. In addition, the body of interest is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The representation of the rotation of ECEF frame from ECI frame is simplified to consider only the constant rotation of the ellipsoid Earth (ω_e) including an initial celestial longitude ($L_G(0)$).



The translational motion of the ECEF coordinate frame is given below, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the body frame. Vre_b is the relative velocity in the wind axes at which the mass flow ($\dot{m}$) is ejected or added to the body in body-fixed axes.

$$\bar{F}_b = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m\left(\dot{\bar{V}}_b + \bar{\omega}_b \times \bar{V}_b + DCM_{bf}\bar{\omega}_e \times \bar{V}_b + DCM_{bf}\left(\bar{\omega}_e \times \left(\bar{\omega}_e \times \bar{X}_f\right)\right)\right)$$

$$+\dot{m}\left(\bar{V}re_b + DCM_{bf}\left(\bar{\omega}_e \times \bar{X}_f\right)\right)$$

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{w}_b \end{bmatrix} = \frac{\bar{F}_b - \dot{m}\left(\bar{V}_{re_b} + DCM_{bf}\left(w_e \times X_f\right)\right)}{m}$$

$$-\left[\bar{\omega}_b \times \bar{V}_b + DCM\bar{\omega}_e \times \bar{V}_b + DCM_{bf}\left(\bar{\omega}_e\left(\bar{\omega}_e \times X_f\right)\right)\right]$$

$$A_{becef} = \frac{\bar{F}_b - \dot{m}\left(\bar{V}_{re_b} + DCM_{bf}\left(\omega_e \times X_f\right)\right)}{m}$$

where the change of position in ECEF $\dot{\bar{x}}_f$ is calculated by

$$\dot{\bar{x}}_f = DCM_{fb}\bar{V}_b$$

and the velocity of the body with respect to ECEF frame, expressed in body frame ($\bar{V}_b$), angular rates of the body with respect to ECI frame, expressed in body frame ($\bar{\omega}_b$). Earth rotation rate ($\bar{\omega}_e$), and relative angular rates of the body with respect to north-east-down (NED) frame, expressed in body frame ($\bar{\omega}_{rel}$) are defined as

$$\bar{V}_b = \begin{bmatrix} u \\ v \\ w \end{bmatrix}, \bar{\omega}_{rel} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \bar{\omega}_e = \begin{bmatrix} 0 \\ 0 \\ \omega_e \end{bmatrix}, \bar{\omega}_b = \bar{\omega}_{rel} + DCM_{bf}\bar{\omega}_e + DCM_{be}\bar{\omega}_{ned}$$

$$\bar{\omega}_{ned} = \begin{bmatrix} \dot{l}\cos\mu \\ -\dot{\mu} \\ -\dot{l}\sin\mu \end{bmatrix} = \begin{bmatrix} V_E/(N+h) \\ -V_N/(M+h) \\ -V_E \bullet \tan\mu/(N+h) \end{bmatrix}$$

The rotational dynamics of the body defined in body-fixed frame are given below, where the applied moments are $[L\ M\ N]^{\mathrm{T}}$, and the inertia tensor I is with respect to the origin O.

$$\bar{M}_b = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = \bar{I}\dot{\bar{\omega}}_b + \bar{\omega}_b \times (\bar{I}\bar{\omega}_b) + \dot{I}\bar{\omega}_b$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The rate of change of the inertia tensor is defined by the following equation.

$$\dot{I} = \begin{bmatrix} \dot{I}_{xx} & -\dot{I}_{xy} & -\dot{I}_{xz} \\ -\dot{I}_{yx} & \dot{I}_{yy} & -\dot{I}_{yz} \\ -\dot{I}_{zx} & -\dot{I}_{zy} & \dot{I}_{zz} \end{bmatrix}$$

The integration of the rate of change of the quaternion vector is given below.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -\frac{1}{2} \begin{bmatrix} 0 & \omega_b(1) & \omega_b(2) & \omega_b(3) \\ -\omega_b(1) & 0 & -\omega_b(3) & \omega_b(2) \\ -\omega_b(2) & \omega_b(3) & 0 & -\omega_b(1) \\ -\omega_b(3) & -\omega_b(2) & \omega_b(1) & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Version History

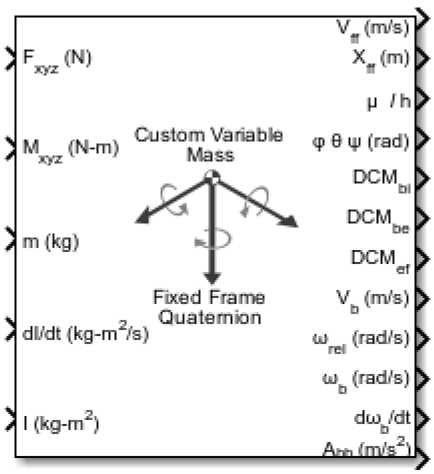
Introduced in R2006a

R2023b: Custom Variable Mass 6DOF ECEF (Quaternion) Block Coordinate Frame Changes

Behavior changed in R2023b

These updates better clarify the coordinate frame of the Custom Variable Mass 6DOF ECEF (Quaternion) block when **Central body** is `Custom`:

- New block icon.



- Port name subscripts of `ecef` have changed to `ff`.
- The **Celestial longitude of Greenwich source** programmatic name has changed from `LG0` to `LPM`. Existing scripts continue to work.
- Updated block parameter names, but models from previous releases continue to work.

Old Parameter Name or Setting	New Parameter Name
ECEF position...	**Fixed frame position**...
When **Planet model** is `Earth`	**Celestial longitude of Greenwich source** and **Celestial longitude of Greenwich [deg]**
When **Planet model** is `Custom`	**Celestial longitude of prime meridian source** and **Celestial longitude of prime meridian [deg]**

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation, 2nd ed*. Hoboken, NJ: John Wiley & Sons, 2003.

[2] McFarland, Richard E. "A Standard Kinematic Model for Flight at NASA-Ames." NASA CR-2497.

[3] "Supplement to Department of Defense World Geodetic System 1984 Technical Report: Part I - Methods, Techniques and Data Used in WGS84 Development" DMA TR8350.2-A.

Extended Capabilities

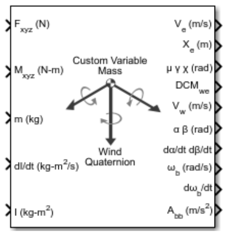
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Custom Variable Mass 6DOF Wind (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion of custom variable mass with respect to wind axes



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Custom Variable Mass 6DOF Wind (Quaternion) block implements a quaternion representation of six-degrees-of-freedom equations of motion of custom variable mass with respect to wind axes. It considers the rotation of a wind-fixed coordinate frame (X_w,Y_w, Z_w) about an flat Earth reference frame (X_e,Y_e, Z_e). The origin of the wind-fixed coordinate frame is the center of gravity of the body. For more information on the wind-fixed coordinate frame, see “Algorithms” on page 5-304.

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Limitations

The block assumes that the applied forces act at the center of gravity of the body.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt — Rates of change of mass
three-element vector

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a three-element vector.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 matrix

Rate of change of inertia tensor matrix, specified as a 3-by-3 matrix.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

I — Inertia tensor matrix
3-by-3 matrix

Inertia tensor matrix, specified as a 3-by-3 matrix.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

$\mathbf{V_{re}}$ — Relative velocities
three-element vector

One or more relative velocities at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

$\mathbf{V_e}$ — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

$\mathbf{X_e}$ — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: double

μ γ χ (rad) — Wind rotation angles
three-element vector

Wind rotation angles [bank, flight path, heading], returned as a three-element vector, in radians.

Data Types: double

DCM_{we} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to wind-fixed axes, returned as a 3-by-3 matrix.

Data Types: double

V_w — Velocity in wind-fixed frame
three-element vector

Velocity in wind-fixed frame, returned as a three-element vector.

Data Types: double

α β (rad) — Angle of attack and sideslip angle
two-element vector

Angle of attack and sideslip angle, returned as a two-element vector, in radians.

Data Types: double

dα/dt dβ/dt — Rate of change of angle of attack and rate of change of sideslip angle
two-element vector

Rate of change of angle of attack and rate of change of sideslip angle, returned as a two-element vector, in radians per second.

Data Types: double

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: double

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: double

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations of the body with respect to the body-fixed axes with the body-fixed coordinate frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this point, select **Include inertial acceleration**.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use

Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Custom Variable` (default) | `Fixed` | `Simple Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Representation — Equations of motion representation

`Quaternion` (default) | `Wind Angles`

Equations of motion representation, specified according to the following table.

`Quaternion`	Use quaternions within equations of motion.
`Wind Angles`	Use wind angles within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-304.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Wind Angles` | `Quaternion`
Default: `'Quaternion'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial airspeed, angle of attack, and sideslip angle [V,alpha,beta] — Initial airspeed, angle of attack, and sideslip angle

`[0 0 0]` (default) | three-element vector

Initial airspeed, angle of attack, and sideslip angle, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial wind orientation [bank angle,flight path angle,heading angle] — Initial wind orientation

`[0 0 0]` (default) | three-element vector

Initial wind angles [bank, flight path, and heading], specified as a three-element vector in radians.

Programmatic Use
Block Parameter: `wind_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. Use state names instead of block paths throughout the linearization process.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state names, specified as a character vector.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

`''` (default) | character vector

Incidence angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector
Values: `''`
Default: `''`

Sideslip angle e.g., 'beta' — Sideslip angle state name

`''` (default) | character vector

Sideslip angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `beta_statename`
Type: character vector
Values: `''`
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

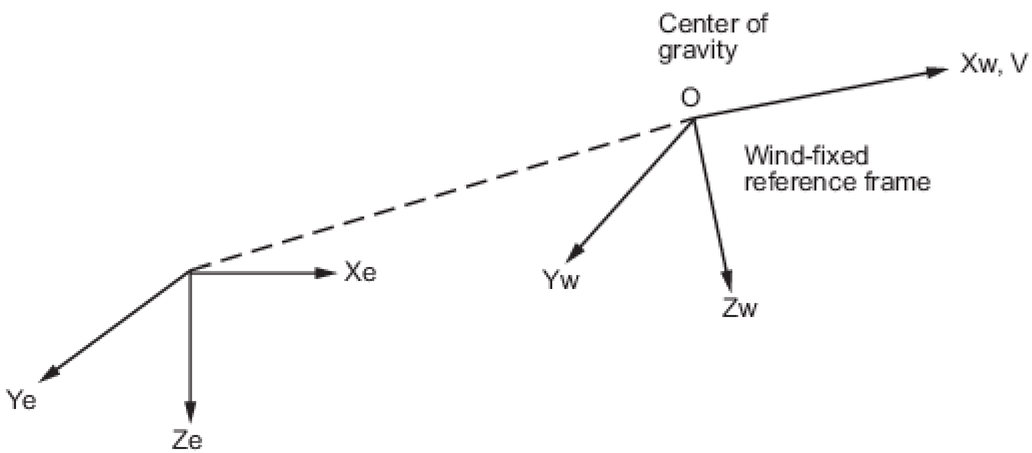
'' (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Algorithms

The origin of the wind-fixed coordinate frame is the center of gravity of the body, and the body is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The flat Earth reference frame is considered inertial, an excellent approximation that allows the forces due to the Earth's motion relative to the “fixed stars” to be neglected.



Flat Earth reference frame

The translational motion of the wind-fixed coordinate frame is given below, where the applied forces $[F_x, F_y, F_z]^T$ are in the wind-fixed frame. Vre_w is the relative velocity in the wind axes at which the mass flow ($\dot{m}$) is ejected or added to the body.

$$\bar{F}_w = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m(\dot{\bar{V}}_w + \bar{\omega}_w \times \bar{V}_w) + \dot{m}\bar{V}re_w$$

$$A_{be} = DCM_{wb}\frac{\left[\bar{F}_w - \dot{m}V_{re}\right]}{m}$$

$$\bar{V}_w = \begin{bmatrix} V \\ 0 \\ 0 \end{bmatrix}, \bar{\omega}_w = \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = DMC_{wb}\begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}, \bar{w}_b = \begin{bmatrix} p_b \\ q_b \\ r_b \end{bmatrix}$$

$$A_{bb} = DCM_{wb}\left[\frac{\bar{F}w - \dot{m}V_{re}}{m} - \bar{\omega}_w \times \bar{V}_w\right]$$

The rotational dynamics of the body-fixed frame are given below, where the applied moments are [*L* *M* *N*]^T, and the inertia tensor *I* is with respect to the origin O. Inertia tensor *I* is easier to define in body-fixed frame.

$$\overline{M}_b = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\overline{\omega}}_b + \overline{\omega}_b \times (I\overline{\omega}_b) + \dot{I}\,\overline{\omega}_b$$

$$A_{bb} = \begin{bmatrix} \dot{U}_b \\ \dot{V}_b \\ \dot{W}_b \end{bmatrix} = DCM_{wb}\left[\frac{\overline{F}w - \dot{m}V_{re}}{m} - \overline{\omega}_w \times \overline{V}_w\right]$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The integration of the rate of change of the quaternion vector is given below.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -\frac{1}{2}\begin{bmatrix} 0 & p & q & r \\ -p & 0 & -r & q \\ -q & r & 0 & -p \\ -r & -q & p & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

C/C++ Code Generation

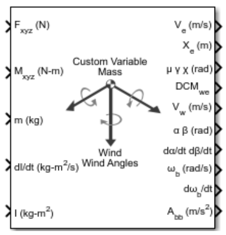
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Custom Variable Mass 6DOF Wind (Wind Angles)

Implement wind angle representation of six-degrees-of-freedom equations of motion of custom variable mass



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Custom Variable Mass 6DOF Wind (Wind Angles) block implements a wind angle representation of six-degrees-of-freedom equations of motion of custom variable mass. For a description of the coordinate system employed and the translational dynamics, see the block description for the Custom Variable Mass 6DOF Wind (Quaternion) block.

For more information of the relationship between the wind angles, see “Algorithms” on page 5-314

Limitations

The block assumes that the applied forces act at the center of gravity of the body.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt — Rates of change of mass
three-element vector

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a three-element vector.

Data Types: `double`

m — Mass
scalar

Mass, specified as a scalar.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 matrix

Rate of change of inertia tensor matrix, specified as a 3-by-3 matrix.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

I — Inertia tensor matrix
3-by-3 matrix

Inertia tensor matrix, specified as a 3-by-3 matrix.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

V_{re} — Relative velocities
three-element vector

One or more relative velocities at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

μ γ χ (rad) — Wind rotation angles
three-element vector

Wind rotation angles [bank, flight path, heading], returned as a three-element vector, in radians.

Data Types: `double`

DCM_{we} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to wind-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_w — Velocity in wind-fixed frame
three-element vector

Velocity in wind-fixed frame, returned as a three-element vector.

Data Types: `double`

α β (rad) — Angle of attack and sideslip angle
two-element vector

Angle of attack and sideslip angle, returned as a two-element vector, in radians.

Data Types: `double`

dα/dt dβ/dt — Rate of change of angle of attack and rate of change of sideslip angle
two-element vector

Rate of change of angle of attack and rate of change of sideslip angle, returned as a two-element vector, in radians per second.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations of the body with respect to the body-fixed axes with the body-fixed coordinate frame, returned as a three-element vector.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this point, select **Include inertial acceleration**.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass type — Mass type

`Custom Variable` (default) | `Simple Variable` | `Fixed`

Mass type, specified according to the following table.

Mass Type	Description	Default for
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Custom Variable` selection conforms to the previously described equations of motion.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Custom Variable'`

Representation — Equations of motion representation

`Wind Angles` (default) | `Quaternion`

Equations of motion representation, specified according to the following table.

Representation	Description
`Wind Angles`	Use Wind angles within equations of motion.
`Quaternion`	Use quaternions within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-314.

Programmatic Use
Block Parameter: rep
Type: character vector
Values: Wind Angles | Quaternion
Default: 'Wind Angles'

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

[0 0 0] (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: xme_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial airspeed, angle of attack, and sideslip angle [V,alpha,beta] — Initial airspeed, angle of attack, and sideslip angle

[0 0 0] (default) | three-element vector

Initial airspeed, angle of attack, and sideslip angle, specified as a three-element vector.

Programmatic Use
Block Parameter: Vm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial wind orientation [bank angle,flight path angle,heading angle] — Initial wind orientation

[0 0 0] (default) | three-element vector

Initial wind angles [bank, flight path, and heading], specified as a three-element vector in radians.

Programmatic Use
Block Parameter: wind_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial body rotation rates [p,q,r] — Initial body rotation

[0 0 0] (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: pm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. Use state names instead of block paths throughout the linearization process.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state names, specified as a character vector.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

`''` (default) | character vector

Incidence angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector
Values: `''`
Default: `''`

Sideslip angle e.g., 'beta' — Sideslip angle state name

`''` (default) | character vector

Sideslip angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `beta_statename`
Type: character vector
Values: `''`
Default: `''`

Wind orientation e.g., {'mu', 'gamma', 'chi'} — Wind orientation state names

`''` (default) | comma-separated list surrounded by braces

Wind orientation state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `wind_statename`
Type: character vector
Values: `''`
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

'' (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Algorithms

The relationship between the wind angles, $[\mu\ \gamma\ \chi]^T$, can be determined by resolving the wind rates into the wind-fixed coordinate frame.

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} \dot{\mu} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\gamma} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} \cos\gamma & 0 & -\sin\gamma \\ 0 & 1 & 0 \\ \sin\gamma & 0 & \cos\gamma \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\chi} \end{bmatrix} \equiv J^{-1} \begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix}$$

Inverting J then gives the required relationship to determine the wind rate vector.

$$\begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix} = J \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} 1 & (\sin\mu\tan\gamma) & (\cos\mu\tan\gamma) \\ 0 & \cos\mu & -\sin\mu \\ 0 & \dfrac{\sin\mu}{\cos\gamma} & \dfrac{\cos\mu}{\cos\gamma} \end{bmatrix} \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix}$$

The body-fixed angular rates are related to the wind-fixed angular rate by the following equation.

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = DMC_{wb} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Using this relationship in the wind rate vector equations, gives the relationship between the wind rate vector and the body-fixed angular rates.

$$\begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix} = J \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} 1 & (\sin\mu\tan\gamma) & (\cos\mu\tan\gamma) \\ 0 & \cos\mu & -\sin\mu \\ 0 & \dfrac{\sin\mu}{\cos\gamma} & \dfrac{\cos\mu}{\cos\gamma} \end{bmatrix} DMC_{wb} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed: Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Delta UT1

Calculate difference between principal Universal Time (UT1) and Coordinated Universal Time (UTC) according to International Astronomical Union (IAU) 2000A reference system

Libraries:
Aerospace Blockset / Environment / Celestial Phenomena
Aerospace Blockset / Utilities / Axes Transformations

Description

The Delta UT1 block calculates the difference between principal UT1 and UTC according to the IAU 2000A reference system and Earth orientation data. By default, this block uses a prepopulated list of International Earth Rotation and Reference Systems Service (IERS) data. This list contains measured and calculated (predicted) data supplied by the IERS. The IERS measures and calculates this data for a set of predetermined dates. For dates after those listed in the prepopulated list, Delta UT1 calculates the data using this equation, limiting the values to +/- .9s:

```
UT1-UTC=0.5309-0.00123(MJD-57808)-(UT2-UT1)
```

Ports

Input

UTC — UT1 for UTC
modified Julian date

UT1 for UTC, specified as a modified Julian date. Use the `mjuliandate` function to convert the UTC date to a modified Julian date.

Data Types: `double`

Output Arguments

ΔUT1 — Difference between UT1 and UTC
double

Difference between UT1 and UTC.

Data Types: `double`

Parameters

IERS data file — Earth orientation data
`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

Programmatic Use
Block Parameter: `FileName`
Type: character vector
Values: `'aeroiersdata.mat'` | MAT-file
Default: `'aeroiersdata.mat'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

IERS data URL — Web site or Earth orientation data file
`https://maia.usno.navy.mil/ser7/finals2000A.data` (default) | web site address | file name

Web site or Earth orientation data file containing the Earth orientation data according to the IAU 2000A, specified as a web site address or file name.

Note If you receive an error message while accessing the default site, use one of these alternate sites:

- `https://datacenter.iers.org/data/latestVersion/10_FINALS.DATA_IAU2000_V2013_0110.txt`
- `ftp://cddis.gsfc.nasa.gov/pub/products/iers/finals2000A.data`

Programmatic Use
Block Parameter: `iersurl`
Type: character vector
Values: `'https://maia.usno.navy.mil/ser7/finals2000A.data'` | web site address | file name
Default: `'https://maia.usno.navy.mil/ser7/finals2000A.data'`

Destination folder — Folder for IERS data file
Current Folder (default)

Folder for IERS data file, specified as a character array or string. Before running this function, create *foldername* with write permission.

To create the IERS data file in the destination folder, click the **Create** button.

Programmatic Use
Block Parameter: `folder`
Type: character vector
Values: `'Current Folder'` | folder name
Default: `'Current Folder'`

Version History

Introduced in R2017b

R2020b: Updated `aeroiersdata.mat` file

Behavior changed in R2020b

The contents of the `aeroiersdata.mat` file have been updated. Correspondingly, the output of this block will have different results when using the default value (`'aeroiersdata.mat'`) as the value of the **IERS data file** parameter. The results reflect more accurate external data from the International Earth Rotation and Reference Systems Service (IERS).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

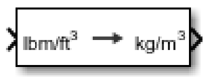
`aeroReadIERSData` | Direction Cosine Matrix ECI to ECEF | Earth Orientation Parameters

Topics

"Calculate UT1 to UTC Values" on page 2-63

Density Conversion

Convert from density units to desired density units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Density Conversion block computes the conversion factor from specified input density units to specified output density units and applies the conversion factor to the input signal.

The Density Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Density
scalar | array

Contains the density, specified as a scalar or array, in initial density units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Density
scalar | array

Contains the density, returned as a scalar or array, in initial density units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`lbm/ft^3` (default) | `kg/m^3` | `slug/ft^3` | `lbm/in^3`

Input units, specified as:

`lbm/ft^3`	Pound mass per cubic foot
`kg/m^3`	Kilograms per cubic meter
`slug/ft^3`	Slugs per cubic foot
`lbm/in^3`	Pound mass per cubic inch

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `'lbm/ft^3'` | `'kg/m^3'` | `'slug/ft^3'` | `'lbm/in^3'`
Default: `'lbm/ft^3'`

Final unit — Output units

`kg/m^3` (default) | `lbm/ft^3` | `slug/ft^3` | `lbm/in^3`

Output units, specified as:

`lbm/ft^3`	Pound mass per cubic foot
`kg/m^3`	Kilograms per cubic meter
`slug/ft^3`	Slugs per cubic foot
`lbm/in^3`	Pound mass per cubic inch

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `'lbm/ft^3'` | `'kg/m^3'` | `'slug/ft^3'` | `'lbm/in^3'`
Default: `'kg/m^3'`

Version History
Introduced before R2006a

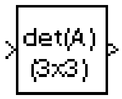
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion

Determinant of 3x3 Matrix

Compute determinant of matrix



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Determinant of 3x3 Matrix block computes the determinant for the input matrix. For related equations, see "Algorithms" on page 5-321.

Ports

Input

Port_1 — Input matrix
3-by-3 matrix

Input matrix, specified as a 3-by-3 matrix.

Data Types: `double`

Output

Port_1 — Determinant
scalar

Determinant, output as a scalar.

Data Types: `double`

Algorithms

The input matrix has the form of

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

The determinant of the matrix has the form of

$$det(A) = A_{11}(A_{22}A_{33} - A_{23}A_{32}) - A_{12}(A_{21}A_{33} - A_{23}A_{31}) + A_{13}(A_{21}A_{32} - A_{22}A_{31})$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

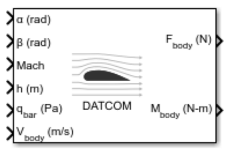
Generate C and C++ code using Simulink® Coder™.

See Also

Adjoint of 3x3 Matrix | Create 3x3 Matrix | Invert 3x3 Matrix

Digital DATCOM Forces and Moments

Compute aerodynamic forces and moments using Digital DATCOM static and dynamic stability derivatives



Libraries:
Aerospace Blockset / Aerodynamics

Description

The Digital DATCOM Forces and Moments block computes the aerodynamic forces and moments about the center of gravity using aerodynamic coefficients from Digital DATCOM.

The Digital DATCOM Forces and Moments block port labels change based on the input and output units selected from the **Units** list.

Limitations

- The Digital DATCOM Forces and Moments block supports only Digital DATCOM, which is the 1976 version of DATCOM.
- The operational limitations of Digital DATCOM apply to the data contained in the **Digital DATCOM structure** parameter. For more information on Digital DATCOM limitations, see Section 2.4.5 of reference [1].
- The **Digital DATCOM structure** parameters `alpha`, `mach`, `alt`, `grndht`, and `delta` must be strictly monotonically increasing to be used with the Digital DATCOM Forces and Moments block.
- The **Digital DATCOM structure** coefficients must correspond to the dimensions of the breakpoints (`alpha`, `mach`, `alt`, `grndht`, and `delta`) to be used with the Digital DATCOM Forces and Moments block.

Ports

Input

Port_1 — Angle of attack
scalar

Angle of attack, specified as a scalar.

Data Types: `double`

Port_2 — Sideslip angle
scalar

Sideslip angle, specified as a scalar, in radians.

Data Types: `double`

Port_3 — Mach number
scalar

Mach number, specified as a scalar.

Data Types: `double`

Port_4 — Altitude
scalar

Altitude, specified as a scalar, in selected length units.

Data Types: `double`

Port_5 — Dynamic pressure
scalar

Dynamic pressure, specified as a scalar, in selected pressure units.

Data Types: `double`

Port_6 — Velocity
three-element vector

Velocity, specified as a three-element vector, in selected velocity units and selected force axes.

Data Types: `double`

Port_7 — Angle of attack rate
scalar

Angle of attack rate, specified as a scalar, in radians per second.

Dependencies

Appears when DAMP Control Card is used in input to Digital DATCOM.

Data Types: `double`

Port_8 — Body angular rates
three-element vector

Body angular rates, specified as a three-element vector, in radians per second.

Dependencies

Appears when DAMP Control Card is used in input to Digital DATCOM.

Data Types: `double`

Port_9 — Ground height
scalar

Ground height, specified as a scalar, in select units of length.

Dependencies

Appears when GRNDEF Namelist is used in input to Digital DATCOM.

Data Types: `double`

Port_10 — Control surface deflection
scalar

Control surface deflection, specified as a scalar, in radians.

Dependencies

Appears when ASYFLP or SYMFLP and GRNDEF namelists are used in input to Digital DATCOM.

Data Types: `double`

Output

Port_1 — Aerodynamic forces at the center of gravity
three-element vector

Aerodynamic forces at the center of gravity, returned as a three-element vector, in selected coordinate system: Body (F, F_{y_x}, and F_z), or Wind (F_D, F_y, and F_L).

Data Types: `double`

Port_2 — Aerodynamic moments at the center of gravity
three-element vector

Aerodynamic moments at the center of gravity, returned as a three-element vector, in body coordinates (M_x, M_y, and M_z).

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Force	Moment	Length	Velocity	Pressure
`Metric (MKS)`	Newton	Newton-meter	Meters	Meters per second	Pascal
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet	Feet per second	Pound per square inch
`English (Velocity in kts)`	Pound	Foot-pound	Feet	Knots	Pound per square inch

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Digital DATCOM structure — Digital DATCOM data structure

`factstruct{1}` (default) | structure

MATLAB structure containing the digital DATCOM data. This structure is generated by the `datcomimport` function. To include dynamic derivatives in the generated output file, call the `datomimport` function with the `damp` keyword.

For more information on creating the digital DATCOM structure, see "Import from USAF Digital DATCOM Files". This example shows how to bring United States Air Force (USAF) Digital DATCOM files into the MATLAB environment using the Aerospace Toolbox software.

Programmatic Use
Block Parameter: `dcase`
Type: character vector
Values: `factstruct{1}` | structure
Default: `factstruct{1}`

Force axes — Coordinate system for aerodynamic force

`Body` (default) | `Wind`

Coordinate system for aerodynamic force, specified as `Body` or `Wind`.

Programmatic Use
Block Parameter: `fmode`
Type: character vector
Values: `'Body'` | `'Wind'`
Default: `'Body'`

Interpolation method — Interpolation method

`None - flat` (default) | `Linear`

Interpolation method, specified as `None (flat)` or `Linear`. The block uses the interpolation method to interpolate the static and dynamic stability coefficients in the **Digital DATCOM structure**.

Programmatic Use
Block Parameter: `imethod`
Type: character vector
Values: `'None (flat)'` | `'Linear'`
Default: `'None (flat)'`

Extrapolation method — Extrapolation method

`None - clip` (default) | `Linear`

Extrapolation method, specified as `None (clip)` or `Linear`. The block uses the extrapolation method to extrapolate the static and dynamic stability coefficients in the **Digital DATCOM structure**.

Programmatic Use
Block Parameter: `emethod`
Type: character vector
Values: `'None (flat)'` | `'Linear'`

Default: `'None (flat)'`

Process out-of-range input — Handle out-of-range input

`Clip to Range` (default) | `Linear Extrapolation`

Handle out-of-range input action, `Linear Extrapolation` or `Clip to Range`.

Programmatic Use
Block Parameter: `rmethod`
Type: character vector
Values: `'Clip to Range'` | `'Linear Extrapolation'`
Default: `'Clip to Range'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Algorithms

Algorithms for calculating forces and moments build up the overall aerodynamic forces and moments ($\boldsymbol{F}$ and $\boldsymbol{M}$) from data contained in the **Digital DATCOM structure** parameter:

$$\boldsymbol{F} = \boldsymbol{F}_{\text{static}} + \boldsymbol{F}_{\text{dyn}} \tag{5-1}$$

$$\boldsymbol{M} = \boldsymbol{M}_{\text{static}} + \boldsymbol{M}_{\text{dyn}} \tag{5-2}$$

$\boldsymbol{F}_{\text{static}}$ and $\boldsymbol{M}_{\text{static}}$ are the static contribution, and $\boldsymbol{F}_{\text{dyn}}$ and $\boldsymbol{M}_{\text{dyn}}$ the dynamic contribution, to the aerodynamic coefficients. If the dynamic characteristics are not contained in the **Digital DATCOM structure** parameter, their contribution is set to zero.

Static Stability Characteristics

Static stability characteristics include the following.

Coefficient	Meaning
C_D	Matrix of drag coefficients. These coefficients are defined positive for an aft-acting load.

Coefficient	Meaning
C_{L}	Matrix of lift coefficients. These coefficients are defined positive for an up-acting load.
C_{m}	Matrix of pitching-moment coefficients. These coefficients are defined positive for a nose-up rotation.
$C_{Y\beta}$	Matrix of derivatives of side-force coefficients with respect to sideslip angle
$C_{n\beta}$	Matrix of derivatives of yawing-moment coefficients with respect to sideslip angle
$C_{l\beta}$	Matrix of derivatives of rolling-moment coefficients with respect to sideslip angle

These are the static contributions to the aerodynamic coefficients in stability axes.

$$C_{D\ static} = C_{D} \tag{5-3}$$

$$C_{y\ static} = C_{Y\beta}\,\beta \tag{5-4}$$

$$C_{L\ static} = C_{L} \tag{5-5}$$

$$C_{l\ static} = C_{l\beta}\,\beta \tag{5-6}$$

$$C_{m\ static} = C_{M} \tag{5-7}$$

$$C_{n\ static} = C_{n\beta}\,\beta \tag{5-8}$$

Dynamic Stability Characteristics

Dynamic stability characteristics include the following.

Coefficient	Meaning
C_{Lq}	Matrix of lift force derivatives due to pitch rate
C_{mq}	Matrix of pitching-moment derivatives due to pitch rate
$C_{Ld\alpha/dt}$	Matrix of lift force derivatives due to rate of angle of attack
$C_{md\alpha/dt}$	Matrix of pitching-moment derivatives due to rate of angle of attack
C_{lp}	Matrix of rolling-moment derivatives due to roll rate
C_{Yp}	Matrix of lateral force derivatives due to roll rate
C_{np}	Matrix of yawing-moment derivatives due to roll rate
C_{nr}	Matrix of yawing-moment derivatives due to yaw rate
C_{lr}	Matrix of rolling-moment derivatives due to yaw rate

These are the dynamic contributions to the aerodynamic coefficients in stability axes.

$$C_{D\ dyn} = 0$$
$$C_{y\ dyn} = C_{yp}p(b_{ref}/2V)$$
$$C_{L\ dyn} = (C_{Lq}q + C_{L\dot{\alpha}}\dot{\alpha})(c_{bar}/2V)$$
$$C_{l\ dyn} = (C_{lp}p + C_{lr}r)(b_{ref}/2V)$$
$$C_{m\ dyn} = (C_{mq}q + C_{m\dot{\alpha}}\dot{\alpha})(c_{bar}/2V)$$
$$C_{n\ dyn} = (C_{np}p + C_{nr}r)(b_{ref}/2V)$$

Version History

Introduced in R2006b

References

[1] *The USAF Stability and Control Digital Datcom*, AFFDL-TR-79-3032, 1979.

[2] Etkin, B., and L. D. Reid. *Dynamics of Flight Stability and Control,* Hoboken, NJ: John Wiley & Sons, 1996.

[3] Roskam, J. "Airplane Design Part VI: Preliminary Calculation of Aerodynamic, Thrust and Power Characteristics", Roskam Aviation and Engineering Corporation, Ottawa, Kansas: 1987.

[4] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

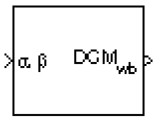
Aerodynamic Forces and Moments | `datcomimport`

Topics

"Import from USAF Digital DATCOM Files"

Direction Cosine Matrix Body to Wind

Convert angle of attack and sideslip angle to direction cosine matrix



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix Body to Wind block converts angle of attack and sideslip angles into a 3-by-3 direction cosine matrix (DCM). This direction cosine matrix is helpful for vector body axes to wind axes coordinate transformations. To transform the coordinates of a vector in body axes (ox_0, oy_0, oz_0) to a vector in wind axes (ox_2, oy_2, oz_2), multiply the block output direction cosine matrix with a vector in body axes. For information on the axis rotations for this transformation, see “Algorithms” on page 5-330.

Ports

Input

α β — Angle of attack and sideslip angle
2-by-1 vector

Angle of attack and sideslip angle, specified as a 2-by-1 vector, in radians.

Data Types: `double`

Output

$\mathbf{DCM_{wb}}$ — Direction cosine matrix
3-by-3 direction cosine matrix

Direction cosine matrix, returned as 3-by-3 direction cosine matrix.

Data Types: `double`

Algorithms

The order of the axis rotations required to bring this transformation about is:

1. A rotation about oy_0 through the angle of attack (α) to axes (ox_1, oy_1, oz_1)
2. A rotation about oz_1 through the sideslip angle (β) to axes (ox_2, oy_2, oz_2)

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = DCM_{wb} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

Combining the two axis transformation matrices defines the following DCM.

$$DCM_{wb} = \begin{bmatrix} \cos\alpha\cos\beta & \sin\beta & \sin\alpha\cos\beta \\ -\cos\alpha\sin\beta & \cos\beta & -\sin\alpha\sin\beta \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation*. Hoboken, NJ: John Wiley & Sons, 1992.

Extended Capabilities

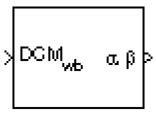
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix Body to Wind to Alpha and Beta | Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Wind Angles | Rotation Angles to Direction Cosine Matrix | Wind Angles to Direction Cosine Matrix

Direction Cosine Matrix Body to Wind to Alpha and Beta

Convert direction cosine matrix to angle of attack and sideslip angle



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix Body to Wind to Alpha and Beta block converts a 3-by-3 direction cosine matrix (DCM) to angle of attack and sideslip angle. The DCM performs the coordinate transformation of a vector in body axes (ox_0, oy_0, oz_0) into a vector in wind axes (ox_2, oy_2, oz_2). For more information on the direction cosine matrix, see “Algorithms” on page 5-333.

Limitations

- This implementation generates angles that lie between ±90 degrees.

Ports

Input

DCM$_{wb}$ — Direction cosine matrix
3-by-3 direction cosine matrix

Direction cosine matrix to transform body-fixed vectors to wind-fixed vectors, specified as a 3-by-3 direct cosine matrix.

Data Types: `double`

Output

α β — Angle of attack and sideslip angle
2-by-1 vector

Angle of attack and sideslip angle, returned as a vector, in radians.

Data Types: `double`

Parameters

Action for invalid DCM — Block behavior
`None` (default) | `Warning` | `Error`

Block behavior when the direction cosine matrix is invalid (not orthogonal).

- `Warning` — Displays warning indicating that the direction cosine matrix is invalid.
- `Error` — Displays error indicating that the direction cosine matrix is invalid.
- `None` — Does not display warning or error (default).

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

Data Types: `char` | `string`

Tolerance for DCM validation — Tolerance

`eps(2)` (default) | scalar

Tolerance of the direction cosine matrix validity, specified as a scalar. The block considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`).
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Programmatic Use
Block Parameter: `tolerance`
Type: character vector
Values: `'eps(2)'` | scalar
Default: `'eps(2)'`

Data Types: `double`

Algorithms

The DCM matrix performs the coordinate transformation of a vector in body axes (ox_0, oy_0, oz_0) into a vector in wind axes (ox_2, oy_2, oz_2). The order of the axis rotations required to bring this about is:

1. A rotation about oy_0 through the angle of attack (α) to axes (ox_1, oy_1, oz_1)
2. A rotation about oz_1 through the sideslip angle (β) to axes (ox_2, oy_2, oz_2)

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = DCM_{wb} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

Combining the two axis transformation matrices defines the following DCM.

$$DCM_{wb} = \begin{bmatrix} \cos\alpha\cos\beta & \sin\beta & \sin\alpha\cos\beta \\ -\cos\alpha\sin\beta & \cos\beta & -\sin\alpha\sin\beta \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

To determine angles from the DCM, the following equations are used:

$$\alpha = \text{asin}(-DCM(3,1))$$

$$\beta = \text{asin}(DCM(1,2))$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

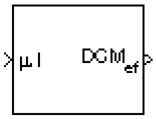
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix Body to Wind | Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Wind Angles | Rotation Angles to Direction Cosine Matrix | Wind Angles to Direction Cosine Matrix

Direction Cosine Matrix ECEF to NED

Convert geodetic latitude and longitude to direction cosine matrix



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix ECEF to NED block converts geodetic latitude and longitude into a 3-by-3 direction cosine matrix (DCM). The DCM matrix performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in north-east-down (NED) axes. For more information on the direction cosine matrix, see “Algorithms” on page 5-335.

The implementation of the ECEF coordinate system assumes that the origin is at the center of the planet, the *x*-axis intersects the Greenwich meridian and the equator, the *z*-axis is the mean spin axis of the planet, positive to the north, and the *y*-axis completes the right-hand system. For more information, see “About Aerospace Coordinate Systems” on page 2-8.

Ports

Input

μl — Geodetic latitude and longitude
2-by-1 vector

Geodetic latitude and longitude, specified as a 2-by-1 vector, in degrees. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `double`

Output

DCM_{ef} — Direction cosine matrix
3-by-3 matrix

DCM to perform coordinate transform of a vector in ECEF axes into a vector in NED axes, returned as a 3-by-3 matrix.

Data Types: `double`

Algorithms

The DCM matrix performs the coordinate transformation of a vector in ECEF axes, (ox_0, oy_0, oz_0), into a vector in NED axes, (ox_2, oy_2, oz_2). The order of the axis rotations required to bring this about is:

1 A rotation about oz_0 through the longitude (ι) to axes (ox_1, oy_1, oz_1)

2 A rotation about oy_1 through the geodetic latitude (μ) to axes (ox_2, oy_2, oz_2)

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = DCM_{ef} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = \begin{bmatrix} -\sin\mu & 0 & \cos\mu \\ 0 & 1 & 0 \\ -\cos\mu & 0 & -\sin\mu \end{bmatrix} \begin{bmatrix} \cos\iota & \sin\iota & 0 \\ -\sin\iota & \cos\iota & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

Combining the two axis transformation matrices defines the following DCM.

$$DCM_{ef} = \begin{bmatrix} -\sin\mu\cos\iota & -\sin\mu\sin\iota & \cos\mu \\ -\sin\iota & \cos\iota & 0 \\ -\cos\mu\cos\iota & -\cos\mu\sin\iota & -\sin\mu \end{bmatrix}$$

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

[2] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. Second Edition. Reston, VA: AIAA Education Series, 2000.

[3] *Recommended Practice for Atmospheric and Space Flight Vehicle Coordinate Systems,* R-004-1992, ANSI/AIAA, February 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

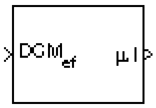
Direction Cosine Matrix ECEF to NED to Latitude and Longitude | Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Wind Angles | ECEF Position to LLA | LLA to ECEF Position | Rotation Angles to Direction Cosine Matrix | Wind Angles to Direction Cosine Matrix

Topics

"About Aerospace Coordinate Systems" on page 2-8

Direction Cosine Matrix ECEF to NED to Latitude and Longitude

Convert direction cosine matrix to geodetic latitude and longitude



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix ECEF to NED to Latitude and Longitude block converts a 3-by-3 direction cosine matrix (DCM) into geodetic latitude and longitude. The DCM matrix performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes, (ox_0, oy_0, oz_0), into geodetic latitude and longitude. For more information on the direction cosine matrix, see “Algorithms” on page 5-338.

Limitations

The DCM matrix performs the coordinate transformation of a vector in ECEF axes, (ox_0, oy_0, oz_0), into geodetic latitude and longitude. The order of the axis rotations required to bring this about is:

- This implementation generates a geodetic latitude that lies between ±90 degrees, and longitude that lies between ±180 degrees.
- The implementation of the ECEF coordinate system assumes that the origin is at the center of the planet, the *x*-axis intersects the Greenwich meridian and the equator, the *z*-axis is the mean spin axis of the planet, positive to the north, and the *y*-axis completes the right-hand system. For more information, see “About Aerospace Coordinate Systems” on page 2-8.

Ports

Input

DCM_{ef} — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix from which to geodetic latitude and longitude, specified as a 3-by-3 matrix.

Data Types: `double`

Output

μl — Geodetic latitude and longitude
2-by-1 vector

Geodetic latitude and longitude, returned as a 2-by-1 vector in degrees.

Data Types: `double`

Parameters

Action for invalid DCM — Block behavior

`None` (default) | `Warning` | `Error`

Block behavior when direction cosine matrix is invalid (not orthogonal).

- `Warning` — Displays warning indicating that the direction cosine matrix is invalid.
- `Error` — Displays error indicating that the direction cosine matrix is invalid.
- `None` — Does not display warning or error (default).

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

Data Types: `char` | `string`

Tolerance for DCM validation — Tolerance

`eps(2)` (default) | scalar

Tolerance of the direction cosine matrix validity, specified as a scalar. The block considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`).
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Programmatic Use
Block Parameter: `tolerance`
Type: character vector
Values: `'eps(2)'` | scalar
Default: `'eps(2)'`

Data Types: `double`

Algorithms

The DCM matrix performs the coordinate transformation of a vector in ECEF axes, (ox_0, oy_0, oz_0), into geodetic latitude and longitude. The order of the axis rotations required to bring this about is:

1. A rotation about oz_0 through the longitude (ι) to axes (ox_1, oy_1, oz_1)
2. A rotation about oy_1 through the geodetic latitude (μ) to axes (ox_2, oy_2, oz_2)

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = DCM_{ef} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

$$\begin{bmatrix} ox_2 \\ oy_2 \\ oz_2 \end{bmatrix} = \begin{bmatrix} -\sin\mu & 0 & \cos\mu \\ 0 & 1 & 0 \\ -\cos\mu & 0 & -\sin\mu \end{bmatrix} \begin{bmatrix} \cos\iota & \sin\iota & 0 \\ -\sin\iota & \cos\iota & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

Combining the two axis transformation matrices defines the following DCM.

$$DCM_{ef} = \begin{bmatrix} -\sin\mu\cos\iota & -\sin\mu\sin\iota & \cos\mu \\ -\sin\iota & \cos\iota & 0 \\ -\cos\mu\cos\iota & -\cos\mu\sin\iota & -\sin\mu \end{bmatrix}$$

To determine geodetic latitude and longitude from the DCM, the following equations are used:

$$\mu = \operatorname{asin}(-DCM(3,3))$$

$$\iota = \operatorname{atan}\left(\frac{-DCM(2,1)}{DCM(2,2)}\right)$$

Version History

Introduced before R2006a

References

[1] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. Second Edition. Reston, VA: AIAA Education Series, 2000.

[2] *Recommended Practice for Atmospheric and Space Flight Vehicle Coordinate Systems,* R-004-1992, ANSI/AIAA, February 1992.

[3] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

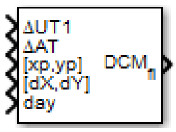
Direction Cosine Matrix ECEF to NED | Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Wind Angles | ECEF Position to LLA | LLA to ECEF Position | Rotation Angles to Direction Cosine Matrix | Wind Angles to Direction Cosine Matrix

Topics

“Algorithms” on page 5-338
“About Aerospace Coordinate Systems” on page 2-8

Direction Cosine Matrix ECI to ECEF

Convert Earth-centered inertial (ECI) to Earth-centered Earth-fixed (ECEF) coordinates



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix ECI to ECEF block calculates the position direction cosine matrix (Earth-centered inertial to Earth-centered Earth-fixed), based on the specified reduction method and Coordinated Universal Time (UTC), for the specified time and geophysical data.

Ports

Input

ΔUT1 — Difference between UTC and Universal Time
scalar

Difference between UTC and Universal Time (UT1) in seconds, specified as a scalar, for which the function calculates the direction cosine or transformation matrix.

Example: `0.234`

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

ΔAT — Difference between International Atomic Time and UTC
scalar

Difference between International Atomic Time (IAT) and UTC, specified as a scalar, in seconds, for which the function calculates the direction cosine or transformation matrix.

Example: 32

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

[xp,yp] — Polar displacement of Earth
1-by-2 array

Polar displacement of Earth, specified as a 1-by-2 array, in radians, from the motion of the Earth crust, along the *x*-axis and *y*-axis.

Example: `[-0.0682e-5 0.1616e-5]`

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

Port_5 — Adjustment based on reduction method
1-by-2 array

Adjustment based on reduction method, specified as 1-by-2 array. The name of the port depends on the setting of the **Reduction** parameter:

- If the reduction method is `IAU-2000/2006`, this input is the adjustment to the location of the Celestial Intermediate Pole (CIP), specified in radians. This location ([dX, dY]) is along the x-axis and y-axis.
- If the reduction method is `IAU-76/FK5`, this input is the adjustment to the longitude ([Δδψ, Δδε]), specified in radians.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`).

Example: `[-0.2530e-6 -0.0188e-6]`

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

Port_6 — Time increment source
scalar

Time increment source, specified as a scalar, such as the Clock block.

Dependencies

- The port name and time increment depend on the **Time Increment** parameter.

Time Increment Value	Port Name
`Day`	**day**
`Hour`	**hour**
`Min`	**min**
`Sec`	**sec**
`None`	No port

- To disable this port, set the **Time Increment** parameter to `None`.

Data Types: `double`

Output

DCM$_{fi}$ — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix ECI to ECEF.

Data Types: `double`

Parameters

Reduction — Reduction method

`IAU-76/FK5` (default) | `IAU-2000/2006`

Reduction method to calculate the direction cosine matrix. The method can be one of the following:

- `IAU-76/FK5`

  Reduce the calculation using the IAU-76/Fifth Fundamental Catalogue (FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, the block calculates the transformation matrix rather than the direction cosine matrix.

- `IAU-2000/2006`

  Reduce the calculation using the International Astronomical Union (IAU)-2000/2006 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation.

Programmatic Use
Block Parameter: `red`
Type: character vector
Values: `'IAU-2000/2006'` | `'IAU-76/FK5'`
Default: `'IAU-2000/2006'`

Year — Year

`2013` (default) | double, whole number, greater than 1

Year to calculate the Coordinated Universal Time (UTC) date. Enter a double value that is a whole number greater than 1, such as `2013`.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: double, whole number, greater than 1
Default: `'2013'`

Month — Month

`January` (default) | `February` | `March` | `April` | `May` | `June` | `July` | `August` | `September` | `October` | `November` | `December`

Month to calculate the UTC date.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Day

1 (default) | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31

Day to calculate the UTC date.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'` | `'10'` | `'11'` | `'12'` | `'13'` | `'14'` | `'15'` | `'16'` | `'17'` | `'18'` | `'19'` | `'20'` | `'21'` | `'22'` | `'23'` | `'24'` | `'25'` | `'26'` | `'27'` | `'28'` | `'29'` | `'30'` | `'31'`
Default: `'1'`

Hour — Hour

`0` (default) | double, whole number, in the range of `0` and `24`

Hour to calculate the UTC date. Enter a double value that is a whole number, from `0` to `24`.

Programmatic Use
Block Parameter: `hour`
Type: character vector
Values: double, whole number, `0` to `24`
Default: `'0'`

Minutes — Minutes

`0` (default) | double, whole number, in the range of `0` and `60`

Minutes to calculate the UTC date. Enter a double value that is a whole number, from `0` to `60`.

Programmatic Use
Block Parameter: `min`
Type: character vector
Values: double, whole number, `0` to `60`
Default: `'0'`

Seconds — Seconds

`0` (default) | double, whole number, in the range of `0` and `60`

Seconds to calculate the UTC date. Enter a double value that is a whole number, from `0` to `60`.

Programmatic Use
Block Parameter: `sec`
Type: character vector

Values: double, whole number, `0` to `60`
Default: `'0'`

Time increment — Time increment

`Day` (default) | `Hour` | `Min` | `Sec` | `None`

Time increment between the specified date and the desired model simulation time. The block adjusts the calculated direction cosine matrix to take into account the time increment from model simulation. For example, selecting `Day` and connecting a simulation timer to the port means that each time increment unit is one day and the block adjusts its calculation based on that simulation time.

This parameter corresponds to the time increment input, the clock source.

If you select `None`, the calculated Julian date does not take into account the model simulation time.

Programmatic Use
Block Parameter: `deltaT`
Type: character vector
Values: `'None'` | `'Day'` | `'Hour'` | `'Min'` | `'Sec'`
Default: `'Day'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as follows.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `errorflag`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`

Higher accuracy parameters — Enable higher accuracy parameters

`on` (default) | `off`

Select this check box to enable these inputs. These inputs let you better control the conversion result. See “Input” on page 5-340 for a description.

- Δ *UT1*
- Δ *AT*
- [*xp* , *yp*]
- [$\Delta\delta\psi$, $\Delta\delta\varepsilon$] or [d *X*, d *Y*]

Programmatic Use
Block Parameter: `extraparamflag`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Version History
Introduced in R2013b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Delta UT1 | Earth Orientation Parameters | LLA to ECEF Position | ECEF Position to LLA | Geocentric to Geodetic Latitude | Geodetic to Geocentric Latitude

Direction Cosine Matrix to Rodrigues

Convert direction cosine matrix to Euler-Rodrigues vector

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix to Rodrigues block determines the 3-by-3 direction cosine matrix from a three-element Euler-Rodrigues vector. The input direction cosine matrix and resulting Euler-Rodrigues vector represent a right-hand passive transformation from frame A to frame B. For more information on the direction cosine matrix, see “Algorithms” on page 5-347.

Ports

Input

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix, from which to determine the Euler-Rodrigues vector.

Data Types: `double`

Output

rod — Euler-Rodrigues vector
three-element vector

Euler-Rodrigues vector, returned as a three-element vector.

Data Types: `double`

Parameters

Action for invalid DCM — Block behavior

`None` (default) | `Warning` | `Error`

Block behavior when direction cosine matrix is invalid (not orthogonal).

- `Warning` — Displays warning and indicates that the direction cosine matrix is invalid.
- `Error` — Displays error and indicates that the direction cosine matrix is invalid.
- `None` — Does not display warning or error (default).

Programmatic Use
Block Parameter: `action`

Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

Data Types: `char` | `string`

Tolerance for DCM validation — Tolerance

`eps(2)` (default) | scalar

Tolerance of direction cosine matrix validity, specified as a scalar. The block considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`)
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Programmatic Use
Block Parameter: `tolerance`
Type: character vector
Values: `'eps(2)'` | scalar
Default: `'eps(2)'`

Data Types: `double`

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x \ b_y \ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

Version History

Introduced in R2017a

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

Extended Capabilities

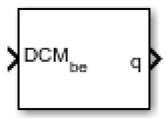
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Rodrigues to Direction Cosine Matrix | Rodrigues to Quaternions | Rodrigues to Rotation Angles | Quaternions to Rodrigues | Rotation Angles to Rodrigues

Direction Cosine Matrix to Quaternions

Convert direction cosine matrix to quaternion vector



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix to Quaternions block transforms a 3-by-3 direction cosine matrix (DCM) into a four-element unit quaternion vector (q_0, q_1, q_2, q_3). Aerospace Blockset uses quaternions that are defined using the scalar-first convention. The direction cosine matrix represents a passive transformation from frame A to frame B. The resulting quaternions represent a series of intrinsic right-hand passive transformations from frame A to frame B. For more information on the direction cosine matrix, see “Algorithms” on page 5-350.

Ports

Input

DCM$_{be}$ — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix to transform the direction cosine matrix to quaternions, specified as a 3-by-3.

Data Types: `double`

Output

q — Quaternion
4-by-1 vector

Quaternion returned by transformation as a 4-by-1 vector.

Data Types: `double`

Parameters

Action for invalid DCM — Block behavior
`None` (default) | `Warning` | `Error`

Block behavior when the direction cosine matrix is invalid (not orthogonal).

- `Warning` — Displays warning indicating that the direction cosine matrix is invalid.
- `Error` — Displays error indicating that the direction cosine matrix is invalid.
- `None` — Does not display warning or error (default).

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

Data Types: `char` | `string`

Tolerance for DCM validation — Tolerance

`eps(2)` (default) | scalar

Tolerance of the direction cosine matrix validity, specified as a scalar. The block considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`).
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Programmatic Use
Block Parameter: `tolerance`
Type: character vector
Values: `'eps(2)'` | scalar
Default: `'eps(2)'`

Data Types: `double`

Algorithms

The DCM is defined as a function of a unit quaternion vector by the following:

$$DCM = \begin{bmatrix} (q_0^2 + q_1^2 - q_2^2 - q_3^2) & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & (q_0^2 - q_1^2 + q_2^2 - q_3^2) & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & (q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix}$$

Using this representation of the DCM, a number of calculations arrive at the correct quaternion. The first of these is to calculate the trace of the DCM to determine which algorithms are used. If the trace is greater than zero, the quaternion can be automatically calculated. When the trace is less than or equal to zero, the major diagonal element of the DCM with the greatest value must be identified to determine the final algorithm used to calculate the quaternion. Once the major diagonal element is identified, the quaternion is calculated.

Version History

Introduced before R2006a

Extended Capabilities

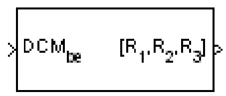
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rotation Angles | Rotation Angles to Direction Cosine Matrix | Rotation Angles to Quaternions | Quaternions to Direction Cosine Matrix | Quaternions to Rotation Angles

Direction Cosine Matrix to Rotation Angles

Convert direction cosine matrix to rotation angles



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix to Rotation Angles block converts a 3-by-3 direction cosine matrix (DCM) into the rotation angles R1, R2, and R3, respectively. The block **Rotation Order** parameter specifies the order of the block output rotations. For example, if **Rotation Order** has a value of ZYX, the block outputs are in the rotation order *z-y-x* (psi theta phi). The direction cosine matrix represents a passive transformation from frame A to frame B. The resulting rotation angles represent a series of right-hand intrinsic passive rotations from frame A to frame B.

Ports

Input

DCM_{be} — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix from which to determine the rotation angles, specified as a 3-by-3 matrix.

Data Types: `double`

Output

$[R_1,R_2,R_3]$ — Rotation angles
3-by-1 vector

Rotation angles, returned as a 3-by-1 vector, in radians.

Data Types: `double`

Parameters

Rotation Order — Block output rotation order

ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation order for three wind rotation angles.

For the ZYX, ZXY, YXZ, YZX, XYZ, and XZY rotations, the block generates an R2 angle that lies between ±pi/2 radians, and R1 and R3 angles that lie between ±pi radians.

For the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` rotations, the block generates an R2 angle that lies between 0 and pi radians, and R1 and R3 angles that lie between ±pi radians. However, in the latter case, R3 is set to 0 radians.

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: `'ZYX'` | `'ZYZ'` |`'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`
Default: `'ZYX'`

Action for invalid DCM — Block behavior
`None` (default) | `Warning` | `Error`

Block behavior when the direction cosine matrix is invalid (not orthogonal).

- `Warning` — Displays warning indicating that the direction cosine matrix is invalid.
- `Error` — Displays error indicating that the direction cosine matrix is invalid.
- `None` — Does not display warning or error (default).

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

Data Types: `char` | `string`

Tolerance for DCM validation — Tolerance

`eps(2)` (default) | scalar

Tolerance of the direction cosine matrix validity, specified as a scalar. The block considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`).
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Programmatic Use
Block Parameter: `tolerance`
Type: character vector
Values: `'eps(2)'` | scalar
Default: `'eps(2)'`

Data Types: `double`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Quaternions | Quaternions to Direction Cosine Matrix | Rotation Angles to Direction Cosine Matrix

Direction Cosine Matrix to Wind Angles

Convert direction cosine matrix to wind angles

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Direction Cosine Matrix to Wind Angles block converts a 3-by-3 direction cosine matrix (DCM) into three wind rotation angles. The DCM matrix performs the coordinate transformation of a vector in earth axes (ox_0, oy_0, oz_0) into a vector in wind axes (ox_3, oy_3, oz_3). For more information on the direction cosine matrix, see “Algorithms” on page 5-356.

This implementation generates a flight path angle that lies between ±90 degrees, and bank and heading angles that lie between ±180 degrees.

Ports

Input

DCM_{we} — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix, to transform Earth-fixed vectors to wind-fixed vectors.

Data Types: `double`

Output

μ γ x — Wind angles
3-by-1 vector

Wind angles (bank, flight path, heading), returned as a 3-by-1 vector, in radians.

Data Types: `double`

Parameters

Action for invalid DCM — Block behavior

`None` (default) | `Warning` | `Error`

Block behavior when the direction cosine matrix is invalid (not orthogonal).

- `Warning` — Displays warning indicating that the direction cosine matrix is invalid.
- `Error` — Displays error indicating that the direction cosine matrix is invalid.

- `None` — Does not display warning or error (default).

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'None'`

Data Types: `char` | `string`

Tolerance for DCM validation — Tolerance

`eps(2)` (default) | scalar

Tolerance of the direction cosine matrix validity, specified as a scalar. The block considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`).
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Programmatic Use
Block Parameter: `tolerance`
Type: character vector
Values: `'eps(2)'` | scalar
Default: `'eps(2)'`

Data Types: `double`

Algorithms

The DCM matrix performs the coordinate transformation of a vector in earth axes (ox_0, oy_0, oz_0) into a vector in wind axes (ox_3, oy_3, oz_3). The order of the axis rotations required to bring this about is:

1. A rotation about oz_0 through the heading angle (χ) to axes (ox_1, oy_1, oz_1)
2. A rotation about oy_1 through the flight path angle (γ) to axes (ox_2, oy_2, oz_2)
3. A rotation about ox_2 through the bank angle (μ) to axes (ox_3, oy_3, oz_3)

$$\begin{bmatrix} ox_3 \\ oy_3 \\ oz_3 \end{bmatrix} = DCM_{we} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

$$\begin{bmatrix} ox_3 \\ oy_3 \\ oz_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} \cos\gamma & 0 & -\sin\gamma \\ 0 & 1 & 0 \\ \sin\gamma & 0 & \cos\gamma \end{bmatrix} \begin{bmatrix} \cos\chi & \sin\chi & 0 \\ -\sin\chi & \cos\chi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

Combining the three axis transformation matrices defines the following DCM.

$$DCM_{we} = \begin{bmatrix} \cos\gamma\cos\chi & \cos\gamma\sin\chi & -\sin\gamma \\ (\sin\mu\sin\gamma\cos\chi - \cos\mu\sin\chi) & (\sin\mu\sin\gamma\sin\chi + \cos\mu\cos\chi) & \sin\mu\cos\gamma \\ (\cos\mu\sin\gamma\cos\chi + \sin\mu\sin\chi) & (\cos\mu\sin\gamma\sin\chi - \sin\mu\cos\chi) & \cos\mu\cos\gamma \end{bmatrix}$$

To determine wind angles from the DCM, the following equations are used:

$$\mu = \operatorname{atan}\left(\frac{DCM(2,3)}{DCM(3,3)}\right)$$

$$\gamma = \operatorname{asin}(-DCM(1,3))$$

$$\chi = \operatorname{atan}\left(\frac{DCM(1,2)}{DCM(1,1)}\right)$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix Body to Wind | Direction Cosine Matrix Body to Wind to Alpha and Beta | Direction Cosine Matrix to Rotation Angles | Rotation Angles to Direction Cosine Matrix | Wind Angles to Direction Cosine Matrix

Discrete Wind Gust Model

Generate discrete wind gust

Libraries:
Aerospace Blockset / Environment / Wind

Description

The Discrete Wind Gust Model block implements a wind gust of the standard "1-cosine" shape. This block implements the mathematical representation in the Military Specification MIL-F-8785C [1]. The gust is applied to each axis individually, or to all three axes at once. You specify the gust amplitude (the increase in wind speed generated by the gust), the gust length (length, in meters, over which the gust builds up) and the gust start time. For more information on the gust shape, see "Algorithms" on page 5-360.

The Discrete Wind Gust Model block can represent the wind speed in units of feet per second, meters per second, or knots.

Ports

Input

V — Air speed
scalar

Airspeed, specified as a scalar, in selected units.

Data Types: `double`

Output

V_{wind} — Wind speed
scalar

Wind speed, returned as a scalar, in selected units.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Units of wind gust, specified as:

Units	Wind	Altitude
`Metric (MKS)`	Meters/second	Meters
`English (Velocity in ft/s)`	Feet/second	Feet
`English (Velocity in kts)`	Knots	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Gust in u-axis — Wind gust to *u*-axis

on (default) | `off`

To apply a wind gust to the *u*-axis in the body frame, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `Gx`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Gust in v-axis — Wind gust to *v*-axis

on (default) | `off`

To apply a wind gust to the *v*-axis in the body frame, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `Gy`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Gust in w-axis — Wind gust to *w*-axis

on (default) | `off`

To apply a wind gust to the *w*-axis in the body frame, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `Gz`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Gust start time (sec) — Gust start time

5 (default) | scalar

Model time, specified as a scalar, at which the gust begins, in seconds.

Programmatic Use
Block Parameter: `t_0`
Type: character vector
Values: scalar
Default: `'5'`

Gust length [dx dy dz] (m) — Gust length

`[120 120 80]` (default)

The length, in meters or feet (depending on the choice of units), over which the gust builds up in each axis. These values must be positive.

Programmatic Use
Block Parameter: `d_m`
Type: character vector
Values: vector
Default: `'[120 120 80]'`

Gust amplitude [ug vg wg] (m/s) — Gust amplitude

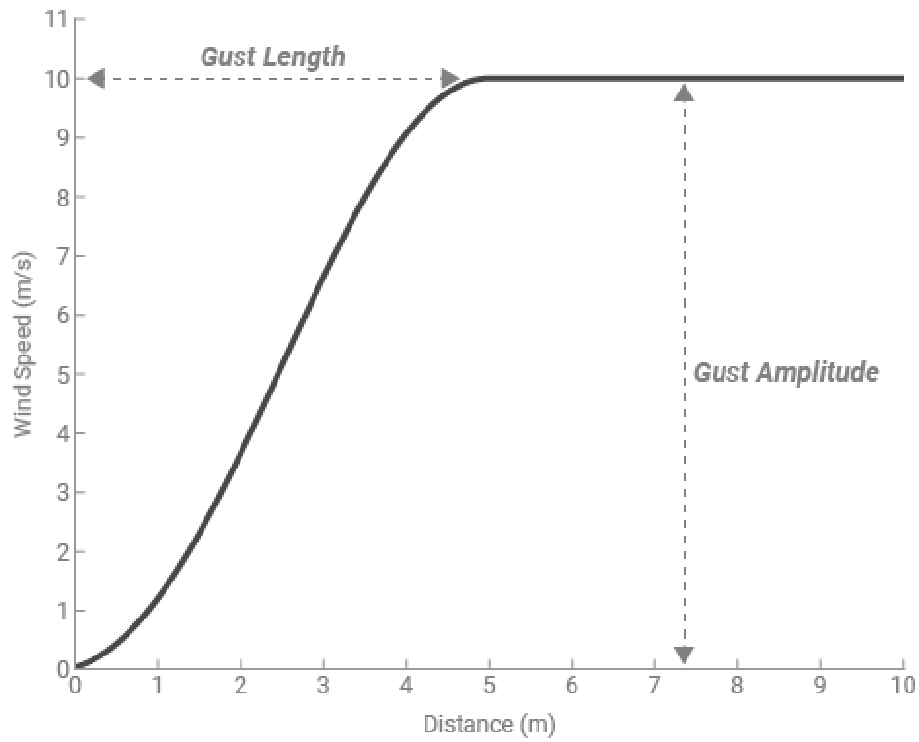
`[3.5 3.5 3.0]` (default)

The magnitude of the increase in wind speed caused by the gust in each axis. These values may be positive or negative.

Programmatic Use
Block Parameter: `d_m`
Type: character vector
Values: vector
Default: `'[3.5 3.5 3.0]'`

Algorithms

This figure shows the shape of the gust with a start time of zero. The parameters that govern the gust shape are indicated on the diagram.



To assess airplane response to large wind disturbances, you can use the discrete gust singly or in multiples.

The mathematical representation of the discrete gust is:

$$V_{wind} = \begin{cases} 0 & x < 0 \\ \frac{V_m}{2}\left(1 - \cos\left(\frac{\pi x}{d_m}\right)\right) & 0 \le x \le d_m \\ V_m & x > d_m \end{cases}$$

where V_m is the gust amplitude, d_m is the gust length, x is the distance traveled, and V_{wind} is the resultant wind velocity in the body axis frame.

Version History

Introduced before R2006a

References

[1] U.S. Military Specification MIL-F-8785C, November 5, 1980.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

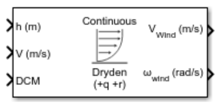
Dryden Wind Turbulence Model (Continuous) | Dryden Wind Turbulence Model (Discrete) | Von Karman Wind Turbulence Model (Continuous)

Topics

"NASA HL-20 Lifting Body Airframe" on page 3-14

Dryden Wind Turbulence Model (Continuous)

Generate continuous wind turbulence with Dryden velocity spectra



Libraries:
Aerospace Blockset / Environment / Wind

Description

The Dryden Wind Turbulence Model (Continuous) block uses the Dryden spectral representation to add turbulence to the aerospace model by passing band-limited white noise through appropriate forming filters. This block implements the mathematical representation in the Military Specification MIL-F-8785C, Military Handbook MIL-HDBK-1797, Military Handbook MIL-HDBK-1797B.

Limitations

The frozen turbulence field assumption is valid for the cases of mean-wind velocity and the root-mean-square turbulence velocity, or intensity, is small relative to the aircraft ground speed.

The turbulence model describes an average of all conditions for clear air turbulence. These factors are not incorporated into the model:

- Terrain roughness
- Lapse rate
- Wind shears
- Mean wind magnitude
- Other meteorological factors

Ports

Input

h — Altitude
scalar

Altitude, specified as a scalar, in selected units.

Data Types: `double`

V — Aircraft speed
scalar

Aircraft speed, specified as a scalar, in selected units.

Data Types: `double`

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix representing the flat Earth coordinates to body-fixed axis coordinates.

Data Types: `double`

Output

V_{wind} — Turbulence velocities
three-element vector

Turbulence velocities, returned as a three-element vector in the same body coordinate reference as the **DCM** input, in specified units.

Data Types: `double`

ω_{wind} — Turbulence angular rates
three-element vector

Turbulence angular rates, specified as a three-element vector, in radians per second.

Data Types: `double`

Parameters

Units — Wind speed units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Units of wind speed due to turbulence, specified as:

Units	Wind Velocity	Altitude	Air Speed
`Metric (MKS)`	Meters/second	Meters	Meters/second
`English (Velocity in ft/s)`	Feet/second	Feet	Feet/second
`English (Velocity in kts)`	Knots	Feet	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Specification — Military reference

`MIL-F-8785C` (default) | `MIL-HDBK-1797` | `MIL-HDBK-1797B`

Military reference, which affects the application of turbulence scale lengths in the lateral and vertical directions, specified as `MIL-F-8785C`, `MIL-HDBK-1797`, or `MIL-HDBK-1797B`.

Programmatic Use
Block Parameter: `spec`
Type: character vector

Values: 'MIL-F-8785C' | 'MIL-HDBK-1797' | 'MIL-HDBK-1797B'
Default: 'MIL-F-8785C'

Model type — Turbulence model
Continuous Dryden (+q -r) (default) | Continuous Von Karman (+q +r) | Continuous Von Karman (-q +r) | Continuous Von Karman (+q -r) | Continuous Dryden (+q +r) | Continuous Dryden (-q +r) | Discrete Dryden (+q -r) | Discrete Dryden (+q +r) | Discrete Dryden (-q +r)

Wind turbulence model, specified as:

Continuous Von Karman (+q -r)	Use continuous representation of Von Kármán velocity spectra with positive vertical and negative lateral angular rates spectra.
Continuous Von Karman (+q +r)	Use continuous representation of Von Kármán velocity spectra with positive vertical and lateral angular rates spectra.
Continuous Von Karman (-q +r)	Use continuous representation of Von Kármán velocity spectra with negative vertical and positive lateral angular rates spectra.
Continuous Dryden (+q -r)	Use continuous representation of Dryden velocity spectra with positive vertical and negative lateral angular rates spectra.
Continuous Dryden (+q +r)	Use continuous representation of Dryden velocity spectra with positive vertical and lateral angular rates spectra.
Continuous Dryden (-q +r)	Use continuous representation of Dryden velocity spectra with negative vertical and positive lateral angular rates spectra.
Discrete Dryden (+q -r)	Use discrete representation of Dryden velocity spectra with positive vertical and negative lateral angular rates spectra.
Discrete Dryden (+q +r)	Use discrete representation of Dryden velocity spectra with positive vertical and lateral angular rates spectra.
Discrete Dryden (-q +r)	Use discrete representation of Dryden velocity spectra with negative vertical and positive lateral angular rates spectra.

The Continuous Dryden selections conform to the transfer function descriptions.

Programmatic Use
Block Parameter: model
Type: character vector
Values: 'Continuous Von Karman (+q +r)' | 'Continuous Von Karman (-q +r)' | 'Continuous Dryden (+q -r)' | 'Continuous Dryden (+q +r)' | 'Continuous Dryden (-q +r)' | 'Discrete Dryden (+q -r)' | 'Discrete Dryden (+q +r)' | 'Discrete Dryden (-q +r)'
Default: 'Continuous Dryden (+q +r)'

Wind speed at 6 m defines the low altitude intensity — Measured wind speed

`15` (default) | real scalar

Measured wind speed at a height of 20 feet (6 meters), specified as a real scalar, which provides the intensity for the low-altitude turbulence model.

Programmatic Use
Block Parameter: `W20`
Type: character vector
Values: real scalar
Default: `'15'`

Wind direction at 6 m (degrees clockwise from north) — Measured wind direction

`0` (default) | real scalar

Measured wind direction at a height of 20 feet (6 meters), specified as a real scalar, which is an angle to aid in transforming the low-altitude turbulence model into a body coordinates.

Programmatic Use
Block Parameter: `Wdeg`
Type: character vector
Values: real scalar
Default: `'0'`

Probability of exceedance of high-altitude intensity — Turbulence intensity

`10^-2 - Light` (default) | `10^-1` | `2x10^-1` | `10^-3 - Moderate` | `10^-4` | `10^-5 - Severe` | `10^-6`

Probability of the turbulence intensity being exceeded, specified as `10^-2 - Light`, `10^-1`, `2x10^-1`, `10^-3 - Moderate`, `10^-4`, `10^-5 - Severe`, or `10^-6`. Above 2000 feet, the turbulence intensity is determined from a lookup table that gives the turbulence intensity as a function of altitude and the probability of the turbulence intensity being exceeded.

Programmatic Use
Block Parameter: `TurbProb`
Type: character vector
Values: `'2x10^-1'` | `'10^-1'` | `'10^-2 - Light'` | `'10^-3 - Moderate'` | `'10^-4'` | `'10^-5 - Severe'` | `'10^-6'`
Default: `'10^-2 - Light'`

Scale length at medium/high altitudes (m) — Turbulence scale length
`533.4` (default) | real scalar

Turbulence scale length above 2000 feet, specified as a real scalar, which is assumed constant. MIL-F-8785C and MIL-HDBK-1797/1797B recommend 1750 feet for the longitudinal turbulence scale length of the Dryden spectra.

Note An alternative scale length value changes the power spectral density asymptote and gust load.

Programmatic Use
Block Parameter: `L_high`
Type: character vector
Values: real scalar
Default: `'533.4'`

Wingspan — Wingspan

`10` (default) | real scalar

Wingspan, specified as a real scalar, which is required in the calculation of the turbulence on the angular rates.

Programmatic Use
Block Parameter: `Wingspan`
Type: character vector
Values: real scalar
Default: `'10'`

Band limited noise sample time (seconds) — Noise sample time

`0.1` (default) | real scalar

Noise sample time, specified as a real scalar, at which the unit variance white noise signal is generated.

Programmatic Use
Block Parameter: `ts`
Type: character vector
Values: real scalar
Default: `'0.1'`

Random noise seeds — Noise seeds [ug vg wg pg]

`[23341 23342 23343 23344]` (default) | four-element vector

Random noise seeds, specified as a four-element vector, which are used to generate the turbulence signals, one for each of the three velocity components and one for the roll rate:

The turbulences on the pitch and yaw angular rates are based on further shaping of the outputs from the shaping filters for the vertical and lateral velocities.

Programmatic Use
Block Parameter: `Seed`
Type: character vector
Values: four-element vector
Default: `'[23341 23342 23343 23344]'`

Turbulence on — Turbulence signals

on (default) | `off`

To generate the turbulence signals, select this check box.

Programmatic Use
Block Parameter: `T_on`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Algorithms

Turbulence is a stochastic process defined by velocity spectra. For an aircraft flying at a speed V through a frozen turbulence field with a spatial frequency of Ω radians per meter, the circular frequency ω is calculated by multiplying V by Ω. MIL-F-8785C and MIL-HDBK-1797/1797B provide these definitions of longitudinal, lateral, and vertical component spectra functions:

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
Longitudinal		
$\Phi_u(\omega)$	$\frac{2\sigma_u^2 L_u}{\pi V} \cdot \frac{1}{1+\left(L_u \frac{\omega}{V}\right)^2}$	$\frac{2\sigma_u^2 L_u}{\pi V} \cdot \frac{1}{1+\left(L_u \frac{\omega}{V}\right)^2}$
$\Phi_{p_g}(\omega)$	$\frac{\sigma_w^2}{V L_w} \cdot \frac{0.8\left(\frac{\pi L_w}{4b}\right)^{1/3}}{1+\left(\frac{4b\omega}{\pi V}\right)^2}$	$\frac{\sigma_w^2}{2V L_w} \cdot \frac{0.8\left(\frac{2\pi L_w}{4b}\right)^{1/3}}{1+\left(\frac{4b\omega}{\pi V}\right)^2}$
Lateral		
$\Phi_v(\omega)$	$\frac{\sigma_v^2 L_v}{\pi V} \cdot \frac{1+3\left(L_v \frac{\omega}{V}\right)^2}{\left[1+\left(L_v \frac{\omega}{V}\right)^2\right]^2}$	$\frac{2\sigma_v^2 L_v}{\pi V} \cdot \frac{1+12\left(L_v \frac{\omega}{V}\right)^2}{\left[1+4\left(L_v \frac{\omega}{V}\right)^2\right]^2}$
$\Phi_r(\omega)$	$\frac{\mp\left(\frac{\omega}{V}\right)^2}{1+\left(\frac{3b\omega}{\pi V}\right)^2} \cdot \Phi_v(\omega)$	$\frac{\mp\left(\frac{\omega}{V}\right)^2}{1+\left(\frac{3b\omega}{\pi V}\right)^2} \cdot \Phi_v(\omega)$
Vertical		
$\Phi_w(\omega)$	$\frac{\sigma_w^2 L_w}{\pi V} \cdot \frac{1+3\left(L_w \frac{\omega}{V}\right)^2}{\left[1+\left(L_w \frac{\omega}{V}\right)^2\right]^2}$	$\frac{2\sigma_w^2 L_w}{\pi V} \cdot \frac{1+12\left(L_w \frac{\omega}{V}\right)^2}{\left[1+4\left(L_w \frac{\omega}{V}\right)^2\right]^2}$
$\Phi_q(\omega)$	$\frac{\pm\left(\frac{\omega}{V}\right)^2}{1+\left(\frac{4b\omega}{\pi V}\right)^2} \cdot \Phi_w(\omega)$	$\frac{\pm\left(\frac{\omega}{V}\right)^2}{1+\left(\frac{4b\omega}{\pi V}\right)^2} \cdot \Phi_w(\omega)$

where:

- b represents the aircraft wingspan.
- L_u, L_v, L_w represent the turbulence scale lengths.
- σ_u, σ_v, σ_w represent the turbulence intensities.

The spectral density definitions of turbulence angular rates are defined in the specifications as three variations:

$p_g = \frac{\partial w_g}{\partial y}$	$q_g = \frac{\partial w_g}{\partial x}$	$r_g = -\frac{\partial v_g}{\partial x}$
$p_g = \frac{\partial w_g}{\partial y}$	$q_g = \frac{\partial w_g}{\partial x}$	$r_g = \frac{\partial v_g}{\partial x}$
$p_g = -\frac{\partial w_g}{\partial y}$	$q_g = -\frac{\partial w_g}{\partial x}$	$r_g = \frac{\partial v_g}{\partial x}$

The variations affect only the vertical (q_g) and lateral (r_g) turbulence angular rates.

The longitudinal turbulence angular rate spectrum,

$$\Phi_{p_g}(\omega)$$

is a rational function. The rational function is derived from curve-fitting a complex algebraic function, not the vertical turbulence velocity spectrum, $\Phi_w(\omega)$, multiplied by a scale factor. The variations exist because the turbulence angular rate spectra contribute less to the aircraft gust response than the turbulence velocity.

The variations result in these combinations of vertical and lateral turbulence angular rate spectra.

Vertical	Lateral
$\Phi_q(\omega)$	$-\Phi_r(\omega)$
$\Phi_q(\omega)$	$\Phi_r(\omega)$
$-\Phi_q(\omega)$	$\Phi_r(\omega)$

To generate a signal with correct characteristics, a band-limited white noise signal is passed through forming filters. The forming filters are derived from the spectral square roots of the spectrum equations.

MIL-F-8785C and MIL-HDBK-1797/1797B provide these transfer functions:

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
Longitudinal		
$H_u(s)$	$\sigma_u\sqrt{\frac{2L_u}{\pi V}}\cdot\frac{1}{1+\frac{L_u}{V}s}$	$\sigma_u\sqrt{\frac{2L_u}{\pi V}}\cdot\frac{1}{1+\frac{L_u}{V}s}$
$H_p(s)$	$\sigma_w\sqrt{\frac{0.8}{V}}\cdot\frac{\left(\frac{\pi}{4b}\right)^{1/6}}{L_w{}^{1/3}\left(1+\left(\frac{4b}{\pi V}\right)s\right)}$	$\sigma_w\sqrt{\frac{0.8}{V}}\cdot\frac{\left(\frac{\pi}{4b}\right)^{1/6}}{(2L_w)^{1/3}\left(1+\left(\frac{4b}{\pi V}\right)s\right)}$
Lateral		
$H_v(s)$	$\sigma_v\sqrt{\frac{L_v}{\pi V}}\cdot\frac{1+\frac{\sqrt{3}L_v}{V}s}{\left(1+\frac{L_v}{V}s\right)^2}$	$\sigma_v\sqrt{\frac{2L_v}{\pi V}}\cdot\frac{1+\frac{2\sqrt{3}L_v}{V}s}{\left(1+\frac{2L_v}{V}s\right)^2}$

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$H_r(s)$	$\dfrac{\mp\frac{s}{V}}{\left(1+\left(\frac{3b}{\pi V}\right)s\right)}\cdot H_v(s)$	$\dfrac{\mp\frac{s}{V}}{\left(1+\left(\frac{3b}{\pi V}\right)s\right)}\cdot H_v(s)$
Vertical		
$H_w(s)$	$\sigma_w\sqrt{\dfrac{L_w}{\pi V}}\cdot\dfrac{1+\frac{\sqrt{3}L_w}{V}s}{\left(1+\frac{L_w}{V}s\right)^2}$	$\sigma_w\sqrt{\dfrac{2L_w}{\pi V}}\cdot\dfrac{1+\frac{2\sqrt{3}L_w}{V}s}{\left(1+\frac{2L_w}{V}s\right)^2}$
$H_q(s)$	$\dfrac{\pm\frac{s}{V}}{\left(1+\left(\frac{4b}{\pi V}\right)s\right)}\cdot H_w(s)$	$\dfrac{\pm\frac{s}{V}}{\left(1+\left(\frac{4b}{\pi V}\right)s\right)}\cdot H_w(s)$

Divided into two distinct regions, the turbulence scale lengths and intensities are functions of altitude.

Note The military specifications result in the same transfer function after evaluating the turbulence scale lengths. The differences in turbulence scale lengths and turbulence transfer functions balance offset.

Low-Altitude Model (Altitude Under 1000 Feet)

According to the military references, the turbulence scale lengths at low altitudes, where *h* is the altitude in feet, are represented in the following table:

MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$L_w = h$ $L_u = L_v = \dfrac{h}{(0.177 + 0.000823h)^{1.2}}$	$2L_w = h$ $L_u = 2L_v = \dfrac{h}{(0.177 + 0.000823h)^{1.2}}$

Typically, at 20 feet (6 meters) the wind speed is 15 knots in light turbulence, 30 knots in moderate turbulence, and 45 knots for severe turbulence. See these turbulence intensities, where W_{20} is the wind speed at 20 feet (6 meters).

$$\sigma_w = 0.1W_{20}$$

$$\frac{\sigma_u}{\sigma_w} = \frac{\sigma_v}{\sigma_w} = \frac{1}{(0.177 + 0.000823h)^{0.4}}$$

The turbulence axes orientation in this region is defined as:

- Longitudinal turbulence velocity, u_g, aligned along the horizontal relative mean wind vector.
- Vertical turbulence velocity, w_g, aligned with vertical.

At this altitude range, the output of the block is transformed into body coordinates.

Medium/High Altitudes (Altitude Above 2000 Feet)

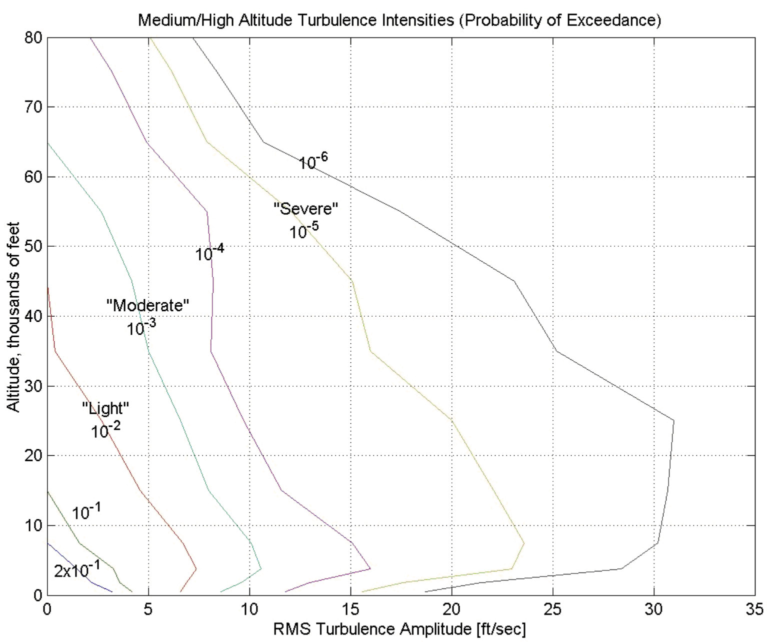
Turbulence scale lengths and intensities for medium-to-high altitudes the are based on the assumption that the turbulence is isotropic. MIL-F-8785C and MIL-HDBK-1797/1797B provide these representations of scale lengths:

MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$L_u = L_v = L_w = 1750$ ft	$L_u = 2L_v = 2L_w = 1750$ ft

The turbulence intensities are determined from a lookup table that provides the turbulence intensity as a function of altitude and the probability of the turbulence intensity being exceeded. The relationship of the turbulence intensities is represented in the following equation:

$\sigma_u = \sigma_v = \sigma_w$.

The turbulence axes orientation in this region is defined as being aligned with the body coordinates.



Between Low and Medium/High Altitudes (Between 1000 and 2000 Feet)

At altitudes between 1000 and 2000, the turbulence velocities and turbulence angular rates are determined by linearly interpolating between the value from the low-altitude model at 1000 feet transformed from mean horizontal wind coordinates to body coordinates and the value from the high-altitude model at 2000 feet in body coordinates.

Version History

Introduced before R2006a

References

[1] Chalk, Charles, T.P. Neal, T.M. Harris, Francis E. Pritchard, and Robert J. Woodcock. Background Information and User Guide for MIL-F-8785B(ASG), "Military Specification-Flying Qualities of Piloted Airplanes." AD869856. Buffalo, NY: Cornell Aeronautical Laboratory, 1969.

[2] *Flying Qualities of Piloted Aircraft*. Department of Defense Handbook. MIL-HDBK-1797. Washington, DC: U.S. Department of Defense, 1997.

[3] *Flying Qualities of Piloted Aircraft*. Department of Defense Handbook. MIL-HDBK-1797B. Washington, DC: U.S. Department of Defense, 2012.

[4] *Flying Qualities of Piloted Airplanes*. U.S. Military Specification MIL-F-8785C. Washington, D.C.: U.S. Department of Defense, 1980.

[5] Hoblit, Frederic M., *Gust Loads on Aircraft: Concepts and Applications*. Reston, VA: AIAA Education Series, 1988.

[6] Ly, U., and Y. Chan. "Time-Domain Computation of Aircraft Gust Covariance Matrices." AIAA Paper 80-1615. Presented at the 6th Atmospheric Flight Mechanics Conference, Danvers, MA, August 1980.

[7] McFarland, Richard E. "A Standard Kinematic Model for Flight Simulation at NASA-Ames." NASA CR-2497. Mountain View, CA: Computer Sciences Corporation, 1975.

[8] McRuer, Duane, Dunstan Graham, and Irving Ashkenas. *Aircraft Dynamics and Automatic Control.* Princeton, NJ: Princeton University Press, 1974, R1990.

[9] Moorhouse, David J., and Robert J. Woodcock. Background Information and User Guide for MIL-F-8785C, "Military Specification—Flying Qualities of Piloted Airplanes." ADA119421. Wright-Patterson AFB, OH: Air Force Wright Aeronautical Labs, 1982.

[10] Tatom, Frank B., George H. Fichtl, and Stephen R. Smith. "Simulation of Atmospheric Turbulent Gusts and Gust Gradients." AIAA Paper 81-0300. Presented at the 19th Aerospace Sciences Meeting, St. Louis, MO, January 1981.

[11] Yeager, Jessie, Implementation and Testing of Turbulence Models for the F18-HARV Simulation. NASA CR-1998-206937. Hampton, VA: Lockheed Martin Engineering & Sciences, 1998.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

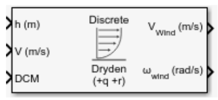
Dryden Wind Turbulence Model (Discrete) | Discrete Wind Gust Model | Von Karman Wind Turbulence Model (Continuous) | Wind Shear Model

Topics

“NASA HL-20 Lifting Body Airframe” on page 3-14

Dryden Wind Turbulence Model (Discrete)

Generate discrete wind turbulence with Dryden velocity spectra



Libraries:
Aerospace Blockset / Environment / Wind

Description

The Dryden Wind Turbulence Model (Discrete) block uses the Dryden spectral representation to add turbulence to the aerospace model by using band-limited white noise with appropriate digital filter finite difference equations. This block implements the mathematical representation in the Military Specification MIL-F-8785C, Military Handbook MIL-HDBK-1797, and Military Handbook MIL-HDBK-1797B. For more information, see “Algorithms” on page 5-378.

Limitations

The frozen turbulence field assumption is valid for the cases of mean-wind velocity and the root-mean-square turbulence velocity, or intensity, is small relative to the aircraft's ground speed.

The turbulence model describes an average of all conditions for clear air turbulence because the following factors are not incorporated into the model:

- Terrain roughness
- Lapse rate
- Wind shears
- Mean wind magnitude
- Other meteorological factions (except altitude)

Ports

Input

h — Altitude
scalar

Altitude, specified as a scalar, in selected units.

Data Types: `double`

V — Aircraft speed
scalar

Aircraft speed, specified as a scalar, in selected units.

Data Types: `double`

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix representing the flat Earth coordinates to body-fixed axis coordinates.

Data Types: `double`

Output

V_{wind} — Turbulence velocities
three-element vector

Turbulence velocities, returned as a three-element vector in the same body coordinate reference as the **DCM** input, in specified units.

Data Types: `double`

ω_{wind} — Turbulence angular rates
three-element vector

Turbulence angular rates, specified as a three-element vector, in radians per second.

Data Types: `double`

Parameters

Units — Wind speed units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Units of wind speed due to turbulence, specified as:

Units	Wind Velocity	Altitude	Air Speed
`Metric (MKS)`	Meters/second	Meters	Meters/second
`English (Velocity in ft/s)`	Feet/second	Feet	Feet/second
`English (Velocity in kts)`	Knots	Feet	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Specification — Military reference

`MIL-F-8785C` (default) | `MIL-HDBK-1797` | `MIL-HDBK-1797B`

Military reference, which affects the application of turbulence scale lengths in the lateral and vertical directions, specified as `MIL-F-8785C`, `MIL-HDBK-1797`, or `MIL-HDBK-1797B`.

Programmatic Use
Block Parameter: `spec`
Type: character vector

Values: 'MIL-F-8785C' | 'MIL-HDBK-1797' | 'MIL-HDBK-1797B'
Default: 'MIL-F-8785C'

Model type — Turbulence model
Discrete Dryden (+q +r) (default) | Continuous Von Karman (+q +r) | Continuous Von Karman (-q +r) | Continuous Dryden (+q -r) | Continuous Dryden (+q +r) | Continuous Dryden (-q +r) | Discrete Dryden (+q -r) | Continuous Von Karman (+q -r) | Discrete Dryden (-q +r)

Select the wind turbulence model to use:

Continuous Von Karman (+q -r)	Use continuous representation of Von Kármán velocity spectra with positive vertical and negative lateral angular rates spectra.
Continuous Von Karman (+q +r)	Use continuous representation of Von Kármán velocity spectra with positive vertical and lateral angular rates spectra.
Continuous Von Karman (-q +r)	Use continuous representation of Von Kármán velocity spectra with negative vertical and positive lateral angular rates spectra.
Continuous Dryden (+q -r)	Use continuous representation of Dryden velocity spectra with positive vertical and negative lateral angular rates spectra.
Continuous Dryden (+q +r)	Use continuous representation of Dryden velocity spectra with positive vertical and lateral angular rates spectra.
Continuous Dryden (-q +r)	Use continuous representation of Dryden velocity spectra with negative vertical and positive lateral angular rates spectra.
Discrete Dryden (+q -r)	Use discrete representation of Dryden velocity spectra with positive vertical and negative lateral angular rates spectra.
Discrete Dryden (+q +r)	Use discrete representation of Dryden velocity spectra with positive vertical and lateral angular rates spectra.
Discrete Dryden (-q +r)	Use discrete representation of Dryden velocity spectra with negative vertical and positive lateral angular rates spectra.

The Discrete Dryden selections conform to the transfer function descriptions.

Programmatic Use
Block Parameter: model
Type: character vector
Values: 'Continuous Von Karman (+q +r)' | 'Continuous Von Karman (-q +r)' | 'Continuous Dryden (+q -r)' | 'Continuous Dryden (+q +r)' | 'Continuous Dryden (-q +r)' | 'Discrete Dryden (+q -r)' | 'Discrete Dryden (+q +r)' | 'Discrete Dryden (-q +r)'
Default: 'Discrete Dryden (+q +r)'

Wind speed at 6 m defines the low altitude intensity — Measured wind speed

15 (default) | real scalar

Measured wind speed at a height of 20 feet (6 meters), specified as a real scalar, which provides the intensity for the low-altitude turbulence model.

Programmatic Use
Block Parameter: W20
Type: character vector
Values: real scalar
Default: '15'

Wind direction at 6 m (degrees clockwise from north) — Measured wind direction

0 (default) | real scalar

Measured wind direction at a height of 20 feet (6 meters), specified as a real scalar, which is an angle to aid in transforming the low-altitude turbulence model into a body coordinates.

Programmatic Use
Block Parameter: Wdeg
Type: character vector
Values: real scalar
Default: '0'

Probability of exceedance of high-altitude intensity — Turbulence intensity

10^-2 - Light (default) | 10^-1 | 2x10^-1 | 10^-3 - Moderate | 10^-4 | 10^-5 - Severe | 10^-6

Probability of the turbulence intensity being exceeded, specified as 10^-2 - Light, 10^-1, 2x10^-1, 10^-3 - Moderate, 10^-4, 10^-5 - Severe, or 10^-6. Above 2000 feet, the turbulence intensity is determined from a lookup table that gives the turbulence intensity as a function of altitude and the probability of the turbulence intensity being exceeded.

Programmatic Use
Block Parameter: TurbProb
Type: character vector
Values: '2x10^-1' | '10^-1' | '10^-2 - Light' | '10^-3 - Moderate' | '10^-4' | '10^-5 - Severe' | '10^-6'
Default: '10^-2 - Light'

Scale length at medium/high altitudes (m) — Turbulence scale length
533.4 (default) | real scalar

Turbulence scale length above 2000 feet, specified as a real scalar, which is assumed constant. From the military references, a figure of 1750 feet is recommended for the longitudinal turbulence scale length of the Dryden spectra.

Note An alternate scale length value changes the power spectral density asymptote and gust load.

Programmatic Use
Block Parameter: `L_high`
Type: character vector
Values: real scalar
Default: `'533.4'`

Wingspan — Wingspan

`10` (default) | real scalar

Wingspan, specified as a real scalar, which is required in the calculation of the turbulence on the angular rates.

Programmatic Use
Block Parameter: `Wingspan`
Type: character vector
Values: real scalar
Default: `'10'`

Band limited noise sample time (seconds) — Noise sample time

`0.1` (default) | real scalar

Noise sample time, specified as a real scalar, at which the unit variance white noise signal is generated.

Programmatic Use
Block Parameter: `ts`
Type: character vector
Values: real scalar
Default: `'0.1'`

Random noise seeds — Noise seeds [ug vg wg pg]

`[23341 23342 23343 23344]` (default) | four-element vector

Random noise seeds, specified as a four-element vector, which are used to generate the turbulence signals, one for each of the three velocity components and one for the roll rate:

The turbulences on the pitch and yaw angular rates are based on further shaping of the outputs from the shaping filters for the vertical and lateral velocities.

Programmatic Use
Block Parameter: `Seed`
Type: character vector
Values: four-element vector
Default: `'[23341 23342 23343 23344]'`

Turbulence on — Turbulence signals

on (default) | `off`

To generate the turbulence signals, select this check box.

Programmatic Use
Block Parameter: `T_on`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Algorithms

According to the military references, turbulence is a stochastic process defined by velocity spectra. For an aircraft flying at a speed *V* through a frozen turbulence field with a spatial frequency of Ω radians per meter, the circular frequency ω is calculated by multiplying *V* by Ω. The following table displays the component spectra functions:

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
Longitudinal		
$\Phi_u(\omega)$	$\frac{2\sigma_u^2 L_u}{\pi V} \cdot \frac{1}{1 + \left(L_u \frac{\omega}{V}\right)^2}$	$\frac{2\sigma_u^2 L_u}{\pi V} \cdot \frac{1}{1 + \left(L_u \frac{\omega}{V}\right)^2}$
$\Phi_p(\omega)$	$\frac{\sigma_w^2}{V L_w} \cdot \frac{0.8\left(\frac{\pi L_w}{4b}\right)^{1/3}}{1 + \left(\frac{4b\omega}{\pi V}\right)^2}$	$\frac{\sigma_w^2}{2V L_w} \cdot \frac{0.8\left(\frac{2\pi L_w}{4b}\right)^{1/3}}{1 + \left(\frac{4b\omega}{\pi V}\right)^2}$
Lateral		
$\Phi_v(\omega)$	$\frac{\sigma_v^2 L_v}{\pi V} \cdot \frac{1 + 3\left(L_v \frac{\omega}{V}\right)^2}{\left[1 + \left(L_v \frac{\omega}{V}\right)^2\right]^2}$	$\frac{2\sigma_v^2 L_v}{\pi V} \cdot \frac{1 + 12\left(L_v \frac{\omega}{V}\right)^2}{\left[1 + 4\left(L_v \frac{\omega}{V}\right)^2\right]^2}$
$\Phi_r(\omega)$	$\frac{\mp\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{3b\omega}{\pi V}\right)^2} \cdot \Phi_v(\omega)$	$\frac{\mp\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{3b\omega}{\pi V}\right)^2} \cdot \Phi_v(\omega)$
Vertical		
$\Phi_w(\omega)$	$\frac{\sigma_w^2 L_w}{\pi V} \cdot \frac{1 + 3\left(L_w \frac{\omega}{V}\right)^2}{\left[1 + \left(L_w \frac{\omega}{V}\right)^2\right]^2}$	$\frac{2\sigma_w^2 L_w}{\pi V} \cdot \frac{1 + 12\left(L_w \frac{\omega}{V}\right)^2}{\left[1 + 4\left(L_w \frac{\omega}{V}\right)^2\right]^2}$
$\Phi_q(\omega)$	$\frac{\pm\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{4b\omega}{\pi V}\right)^2} \cdot \Phi_w(\omega)$	$\frac{\pm\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{4b\omega}{\pi V}\right)^2} \cdot \Phi_w(\omega)$

The variable *b* represents the aircraft wingspan. The variables L_u, L_v, L_w represent the turbulence scale lengths. The variables σ_u, σ_v, σ_w represent the turbulence intensities.

The spectral density definitions of turbulence angular rates are defined in the references as three variations, which are displayed in the following table:

$p_g = \frac{\partial w_g}{\partial y}$	$q_g = \frac{\partial w_g}{\partial x}$	$r_g = -\frac{\partial v_g}{\partial x}$
$p_g = \frac{\partial w_g}{\partial y}$	$q_g = \frac{\partial w_g}{\partial x}$	$r_g = \frac{\partial v_g}{\partial x}$
$p_g = -\frac{\partial w_g}{\partial y}$	$q_g = -\frac{\partial w_g}{\partial x}$	$r_g = \frac{\partial v_g}{\partial x}$

The variations affect only the vertical (q_g) and lateral (r_g) turbulence angular rates.

Keep in mind that the longitudinal turbulence angular rate spectrum, $\Phi_p(\omega)$, is a rational function. The rational function is derived from curve-fitting a complex algebraic function, not the vertical turbulence velocity spectrum, $\Phi_w(\omega)$, multiplied by a scale factor. Because the turbulence angular rate spectra contribute less to the aircraft gust response than the turbulence velocity spectra, it may explain the variations in their definitions.

The variations lead to the following combinations of vertical and lateral turbulence angular rate spectra:

Vertical	Lateral
$\Phi_q(\omega)$	$-\Phi_r(\omega)$
$\Phi_q(\omega)$	$\Phi_r(\omega)$
$-\Phi_q(\omega)$	$\Phi_r(\omega)$

To generate a signal with the correct characteristics, a unit variance, band-limited white noise signal is used in the digital filter finite difference equations.

The following table displays the digital filter finite difference equations:

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
Longitudinal		
u_g	$\left(1-\frac{V}{L_u}T\right)u_g+\sqrt{2\frac{V}{L_u}T}\frac{\sigma_u}{\sigma_\eta}\eta_1$	$\left(1-\frac{V}{L_u}T\right)u_g+\sqrt{2\frac{V}{L_u}T}\frac{\sigma_u}{\sigma_\eta}\eta_1$
p_g	$\left(1-\frac{2.6}{\sqrt{L_w b}}T\right)p_g+$ $\left(\sqrt{2\frac{2.6}{\sqrt{L_w b}}T}\right)\left(\frac{0.95}{\frac{\sqrt[3]{2L_w b^2}}{\sigma_\eta}\eta_4}\right)\sigma_w$	MIL-HDBK-1797 $\left(1-\frac{2.6}{\sqrt{2L_w b}}T\right)p_g+$ $\left(\sqrt{2\frac{2.6}{\sqrt{2L_w b}}T}\right)\left(\frac{1.9}{\frac{\sqrt{2L_w b}}{\sigma_\eta}\eta_4}\right)\sigma_w$

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
		MIL-HDBK-1797B $\left(1-\frac{2.6V}{\sqrt{2L_w b}}T\right)p_g +$ $\left(\sqrt{2\frac{2.6V}{\sqrt{2L_w b}}T}\right)\left(\frac{1.9}{\frac{\sqrt{2L_w b}}{\sigma_\eta}\eta_4}\right)\sigma_w$
Lateral		
v_g	$\left(1-\frac{V}{L_v}T\right)v_g + \sqrt{2\frac{V}{L_v}T}\frac{\sigma_v}{\sigma_\eta}\eta_2$	$\left(1-\frac{V}{L_v}T\right)v_g + \sqrt{2\frac{V}{L_v}T}\frac{\sigma_v}{\sigma_\eta}\eta_2$
r_g	$\left(1-\frac{\pi V}{3b}T\right)r_g \mp \frac{\pi}{3b}\left(v_g - v_{g_{past}}\right)$	$\left(1-\frac{\pi V}{3b}T\right)r_g \mp \frac{\pi}{3b}\left(v_g - v_{g_{past}}\right)$
Vertical		
w_g	$\left(1-\frac{V}{L_w}T\right)w_g + \sqrt{2\frac{V}{L_w}T}\frac{\sigma_w}{\sigma_\eta}\eta_3$	$\left(1-\frac{V}{L_w}T\right)w_g + \sqrt{2\frac{V}{L_w}T}\frac{\sigma_w}{\sigma_\eta}\eta_3$
q_g	$\left(1-\frac{\pi V}{4b}T\right)q_g \pm \frac{\pi}{4b}\left(w_g - w_{g_{past}}\right)$	$\left(1-\frac{\pi V}{4b}T\right)q_g \pm \frac{\pi}{4b}\left(w_g - w_{g_{past}}\right)$

Divided into two distinct regions, the turbulence scale lengths and intensities are functions of altitude.

Low-Altitude Model (Altitude < 1000 feet)

According to the military references, the turbulence scale lengths at low altitudes, where h is the altitude in feet, are represented in the following table:

MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$L_w = h$ $L_u = L_v = \frac{h}{(0.177 + 0.000823h)^{1.2}}$	$2L_w = h$ $L_u = 2L_v = \frac{h}{(0.177 + 0.000823h)^{1.2}}$

The turbulence intensities are given below, where W_{20} is the wind speed at 20 feet (6 m). Typically for light turbulence, the wind speed at 20 feet is 15 knots; for moderate turbulence, the wind speed is 30 knots, and for severe turbulence, the wind speed is 45 knots.

$$\sigma_w = 0.1W_{20}$$

$$\frac{\sigma_u}{\sigma_w} = \frac{\sigma_v}{\sigma_w} = \frac{1}{(0.177 + 0.000823h)^{0.4}}$$

The turbulence axes orientation in this region is defined as follows:

- Longitudinal turbulence velocity, u_g, aligned along the horizontal relative mean wind vector
- Vertical turbulence velocity, w_g, aligned with vertical.

At this altitude range, the output of the block is transformed into body coordinates.

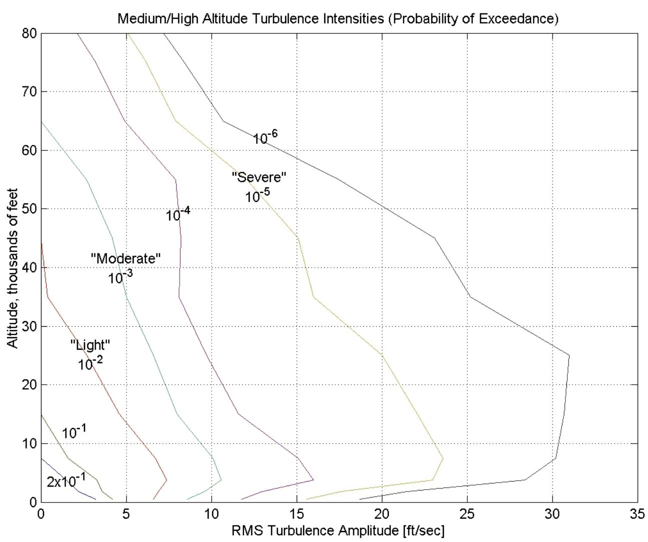
Medium/High Altitudes (Altitude > 2000 feet)

For medium to high altitudes the turbulence scale lengths and intensities are based on the assumption that the turbulence is isotropic. In the military references, the scale lengths are represented by the following equations:

MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$L_u = L_v = L_w = 1750$ ft	$L_u = 2L_v = 2L_w = 1750$ ft

The turbulence intensities are determined from a lookup table that provides the turbulence intensity as a function of altitude and the probability of the turbulence intensity being exceeded. The relationship of the turbulence intensities is represented in the following equation: $\sigma_u = \sigma_v = \sigma_w$.

The turbulence axes orientation in this region is defined as being aligned with the body coordinates.



Between Low and Medium/High Altitudes (1000 feet < Altitude < 2000 feet)

At altitudes between 1000 feet and 2000 feet, the turbulence velocities and turbulence angular rates are determined by linearly interpolating between the value from the low altitude model at 1000 feet transformed from mean horizontal wind coordinates to body coordinates and the value from the high altitude model at 2000 feet in body coordinates.

Version History

Introduced before R2006a

References

[1] U.S. Military Handbook MIL-HDBK-1797B, April 9, 2012.

[2] U.S. Military Handbook MIL-HDBK-1797, December 19, 1997.

[3] U.S. Military Specification MIL-F-8785C, November 5, 1980.

[4] Chalk, Charles, T.P. Neal, T.M. Harris, Francis E. Pritchard, and Robert J. Woodcock. "Background Information and User Guide for MIL-F-8785B(ASG), Military Specification-Flying Qualities of Piloted Airplanes." AD869856. Buffalo, NY: Cornell Aeronautical Laboratory, August 1969.

[5] Hoblit, Frederic M., *Gust Loads on Aircraft: Concepts and Applications*. Reston, VA: AIAA Education Series, 1988.

[6] Ly, U., Chan, Y. "Time-Domain Computation of Aircraft Gust Covariance Matrices," AIAA Paper 80-1615. Presented at the Atmospheric Flight Mechanics Conference, Danvers, Massachusetts, August 11-13, 1980.

[7] McRuer, D., Ashkenas, I., Graham, D., *Aircraft Dynamics and Automatic Control*. Princeton: Princeton University Press, July 1990.

[8] Moorhouse, David J. and Robert J. Woodcock. "Background Information and User Guide for MIL-F-8785C, `Military Specification-Flying Qualities of Piloted Airplanes'." ADA119421, Flight Dynamic Laboratory, July 1982.

[9] McFarland, R. "A Standard Kinematic Model for Flight Simulation at NASA-Ames." NASA CR-2497. Computer Sciences Corporation, January 1975.

[10] Tatom, Frank B., Stephen R. Smith, and George H. Fichtl. "Simulation of Atmospheric Turbulent Gusts and Gust Gradients." AIAA Paper 81-0300, Aerospace Sciences Meeting, St. Louis, MO, January 12-15, 1981.

[11] Yeager, Jessie, "Implementation and Testing of Turbulence Models for the F18-HARV Simulation." NASA CR-1998-206937. Hampton, VA: Lockheed Martin Engineering & Sciences, March 1998.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

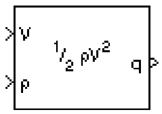
See Also

Dryden Wind Turbulence Model (Continuous) | Discrete Wind Gust Model | Von Karman Wind Turbulence Model (Continuous) | Wind Shear Model

Topics
"NASA HL-20 Lifting Body Airframe" on page 3-14

Dynamic Pressure

Compute dynamic pressure using velocity and air density



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Dynamic Pressure block computes dynamic pressure.

Dynamic pressure is defined as:

$$\bar{q} = \frac{1}{2}\rho V^2,$$

where ρ is air density and V is velocity.

Ports

Input

V — Velocity
three-element vector

Velocity, specified as a three-element vector.

Data Types: `double`

ρ — Air density
scalar

Air density, specified as a scalar.

Data Types: `double`

Output

Output 1 — Dynamic pressure
scalar

Dynamic pressure, returned as a scalar.

Data Types: `double`

Version History

Introduced before R2006a

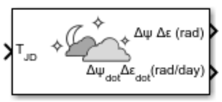
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Aerodynamic Forces and Moments | Mach Number

Earth Nutation

Implement Earth nutation



Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Description

The Earth Nutation block implements the International Astronomical Union (IAU) 1980 nutation series for a given Julian date The block uses the Chebyshev coefficients that the NASA Jet Propulsion Laboratory provides.

The **Epoch** parameter controls the number of block inputs. If you select `Julian date`, the block has one input port, if you select `T0 and elapsed Julian time`, the block has two input ports.

Tip For T_{JD}, Julian date input for the block:

- Calculate the date using the Julian Date Conversion block or the Aerospace Toolbox `juliandate` function.
- Calculate the Julian date using some other means and input it using the Constant block.

Ports

Input

T_{JD} — Julian date
scalar | positive | between minimum and maximum Julian dates

Julian date, specified as a positive scalar between minimum and maximum Julian dates.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `Julian date`.

Data Types: `double`

$T0_{JD}$ — Fixed Julian date
scalar | positive

Fixed Julian date for a specific epoch that is the most recent midnight at or before the interpolation epoch, specified as a positive scalar. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian dates.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `T0 and elapsed Julian time`.

Data Types: `double`

ΔT_{JD} — Elapsed Julian time
scalar | positive

Elapsed Julian time between the fixed Julian date and the ephemeris time, specified as a positive scalar. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian date.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `T0 and elapsed Julian time`.

Data Types: `double`

Output

Δψ Δε (rad) — Earth nutation
vector

Earth nutation, output as a vector of longitude (Δψ) and obliquity (Δε), in rad.

Data Types: `double`

$\Delta\psi_{dot}$ $\Delta\varepsilon_{dot}$ (rad/day) — Earth nutation angular rate
scalar

Earth nutation angular rate for the longitude ($\Delta\psi_{dot}$) and obliquity ($\Delta\varepsilon_{dot}$), specified as a scalar in rad/day.

Dependencies

This port displays if the **Calculate rates** parameter is selected.

Data Types: `double`

Parameters

Epoch — Epoch

`Julian date` (default) | `T0 and elapsed Julian time`

Epoch, specified as:

- `Julian date`

  Julian date to calculate the Earth nutation. When this option is selected, the block has one input port, T_{JD}.
- `T0 and elapsed Julian time`

  Julian date, specified by two block inputs:

- Fixed Julian date representing a starting epoch.
- Elapsed Julian time between the $T0_{JD}$ and the desired model simulation time. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian dates.

Programmatic Use
Block Parameter: `epochflag`
Type: character vector
Values: `Julian date` | `T0 and elapsed Julian time`
Default: `'Julian date'`

Ephemeris model — Ephemeris model

`DE405` (default) | `DE421` | `DE423` | `DE430`

Select an Ephemeris model from the list defined by the Jet Propulsion Laboratory:

Ephemeris Model	Description
`DE405`	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
`DE421`	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
`DE423`	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
`DE430`	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`.

Programmatic Use
Block Parameter: `de`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as follows.

Action	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: errorflag
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'Error'

Calculate rates — Calculate rate of Earth nutation

on (default) | off

Calculate the rate of the Earth nutation by selecting this check box.

Dependencies

Select this check box to display the $\Delta\psi_{dot}$ $\Delta\varepsilon_{dot}$ port.

Programmatic Use
Block Parameter: velflag
Type: character vector
Values: 'off' | 'on' |
Default: 'on'

Version History

Introduced in R2013a

References

[1] Folkner, W. M., J. G. Williams, D. H. Boggs. "The Planetary and Lunar Ephemeris DE 421." *IPN Progress Report* 42-178, 2009.

[2] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, McGraw-Hill, New York, 1997.

Extended Capabilities

C/C++ Code Generation

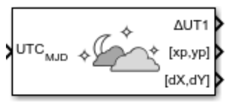
Generate C and C++ code using Simulink® Coder™.

See Also

aeroDataPackage | Moon Libration | Planetary Ephemeris

Earth Orientation Parameters

Calculate Earth orientation parameters (EOP)



Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Description

The Earth Orientation Parameters block calculates these parameters:

- Difference between the UTC and Universal Time (UT1)
- Movement of the rotation axis with respect to the crust of the Earth
- Adjustment to the location of the Celestial Intermediate Pole (CIP)

By default, this block uses a prepopulated list of International Earth Rotation and Reference Systems Service (IERS) data. This list contains measured and calculated (predicted) data supplied by the IERS. The IERS measures and calculates this data for a set of predetermined dates. For dates after those listed in the prepopulated list, Earth Orientation Parameters calculates the ΔUT1 using this equation, limiting the values to +/- 0.9 s:

```
UT1-UTC=0.5309-0.00123(MJD-57808)-(UT2-UT1)
```

Use this block when your application uses Earth Centered Inertial to Earth Centered Earth Fixed transformations, such as for high altitude applications.

Ports

Input

UTC_{MJD} — UT1 for UTC
scalar

UT1 for UTC, specified as a scalar modified Julian date. Use the Julian Date Conversion block to convert the UTC date to a modified Julian date.

Data Types: `double`

Output

ΔUT1 — Difference between UT1 and UTC
scalar

Difference between UT1 and UTC, specified as a scalar, in seconds.

Data Types: `double`

[xp,yp] — Polar displacement of Earth
vector

Polar displacement of the Earth, [*xp*,*yp*], specified as a vector, in radians, from the motion of the Earth crust, along the *x*- and *y*-axes.

Data Types: `double`

[dX,dY] — Adjustment to location of Celestial Intermediate Pole (CIP)
vector

Adjustment to the location of the Celestial Intermediate Pole (CIP), specified as a vector, in radians. This location ([d*X*,d*Y*]) is along the *x*- and *y*-axes.

Data Types: `double`

$\Delta UT1_{err}$ — Return errors for the measured and predicted values in the IERS data
vector

Return errors for the measured and predicted values in the IERS data for the difference between UT1 and UTC, specified as a vector, in seconds.

Dependencies

This port is enabled when the **Output parameter error** is selected.

$[xp,yp]_{err}$ — Return errors for the measured and predicted values in the IERS data
vector

Return errors for the measured and predicted values in the IERS data for the polar displacement of Earth, specified as a vector, in radians.

Dependencies

This port is enabled when the **Output parameter error** is selected.

$[dX,dY]_{err}$ — Return errors for the measured and predicted values in the IERS data
vector

Return errors for the measured and predicted values in the IERS data for the adjustment to location of Celestial Intermediate Pole (CIP), specified as a vector, in radians.

Dependencies

This port is enabled when the **Output parameter error** is selected.

Parameters

IERS data file — Earth orientation data
`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

Programmatic Use

Block Parameter: `FileName`
Type: character vector
Values: scalar
Default: `'aeroiersdata.mat'`

Output parameter error — Enable output ports to return errors
`off` (default) | `on`

Select this parameter to enable output ports to return errors for the measured and predicted values in the IERS data file:

- Difference between UT1 and UTC
- Polar displacement of Earth
- Adjustment to location of Celestial Intermediate Pole (CIP)

Dependencies

Selecting this check box enables these ports:

- $\Delta UT1_{err}$
- $[xp,yp]_{err}$
- $[dX,dY]_{err}$

Programmatic Use
Block Parameter: `OutputError`
Type: character vector
Values: scalar
Default: `'off'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

IERS data URL — Web site or Earth orientation data file
`http://maia.usno.navy.mil/ser7/finals2000A.data` (default) | web site address | file name

Web site or Earth orientation data file containing the Earth orientation data according to the IAU 2000A, specified as a web site address or file name.

Note If you receive an error message while accessing the default site, use one of these alternate sites:

- `https://datacenter.iers.org/data/latestVersion/10_FINALS.DATA_IAU2000_V2013_0110.txt`
- `ftp://cddis.gsfc.nasa.gov/pub/products/iers/finals2000A.data`

Programmatic Use
Block Parameter: `FileName`
Type: character vector
Values: scalar
Default: `'aeroiersdata.mat'`

Destination folder — Folder for IERS data file
current Folder (default)

Folder for IERS data file, specified as a character array or string. Before running this function, create *foldername* with write permission.

To create the IERS data file in the destination folder, click the **Create** button.

Programmatic Use
Block Parameter: `FileName`
Type: character vector
Values: scalar
Default: `'aeroiersdata.mat'`

Version History

Introduced in R2018b

R2020b: Updated `aeroiersdata.mat` file

Behavior changed in R2020b

The contents of the `aeroiersdata.mat` file have been updated. Correspondingly, the output of this block will have different results when using the default value (`'aeroiersdata.mat'`) as the value of the **IERS data file** parameter. The results reflect more accurate external data from the International Earth Rotation and Reference Systems Service (IERS).

Extended Capabilities

C/C++ Code Generation

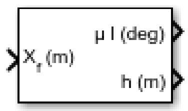
Generate C and C++ code using Simulink® Coder™.

See Also

`aeroReadIERSData` | Delta UT1 | Direction Cosine Matrix ECI to ECEF

ECEF Position to LLA

Calculate geodetic latitude, longitude, and altitude above planetary ellipsoid from Earth-centered Earth-fixed (ECEF) position



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The ECEF Position to LLA block converts a 3-by-1 vector of ECEF position ($\bar{p}$) into geodetic latitude ($\bar{\mu}$), longitude ($\bar{\iota}$), and altitude $\left(\bar{h}\right)$ above the planetary ellipsoid. For more information on the ECEF position, see “Algorithms” on page 5-395.

Limitations

- This implementation generates a geodetic latitude that lies between ±90 degrees, and longitude that lies between ±180 degrees. The planet is assumed to be ellipsoidal. By setting the flattening to 0, you model a spherical planet.
- The implementation of the ECEF coordinate system assumes that its origin lies at the center of the planet, the *x*-axis intersects the prime (Greenwich) meridian and the equator, the *z*-axis is the mean spin axis of the planet (positive to the north), and the *y*-axis completes the right-handed system.

Ports

Input

X_f — Position
3-by-1 vector

Position in ECEF frame, specified as a 3-by-1 vector.

Data Types: `double`

Output

μ l — Geodetic latitude and longitude
2-by-1 vector

Geodetic latitude and longitude, returned as a 2-by-1 vector, in degrees.

Data Types: `double`

h — Altitude
scalar

Altitude above the planetary ellipsoid, returned as a scalar, in the same units as the ECEF position.

Data Types: `double`

Parameters

Units — Output units

`Metric (MKS)` (default) | `English`

Output units, specified as:

Units	Position	Equatorial Radius	Altitude
`Metric (MKS)`	Meters	Meters	Meters
`English`	Feet	Feet	Feet

Dependencies

To enable this parameter, set **Planet model** to `Earth (WGS84)`.

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `'1/298.257223563'`

Equatorial radius of planet — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator, specified as a double scalar, in the same units as the desired units for the ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use

Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `'6378137'`

Algorithms

The ECEF position is defined as:

$$\bar{p} = \begin{bmatrix} \bar{p}_x \\ \bar{p}_y \\ \bar{p}_z \end{bmatrix}.$$

Longitude is calculated from the ECEF position by

$$\iota = \mathrm{atan}\left(\frac{p_y}{p_x}\right).$$

Geodetic latitude ($\bar{\mu}$) is calculated from the ECEF position using Bowring's method, which typically converges after two or three iterations. The method begins with an initial guess for geodetic latitude ($\bar{\mu}$) and reduced latitude $\left(\bar{\beta}\right)$. An initial guess takes the form:

$$\bar{\beta} = \mathrm{atan}\left(\frac{p_z}{(1-f)s}\right)$$

$$\bar{\mu} = \mathrm{atan}\left(\frac{p_z + \frac{e^2(1-f)}{(1-e^2)}R(\sin\beta)^3}{s - e^2 R(\cos\beta)^3}\right)$$

where R is the equatorial radius, f is the flattening of the planet, $e^2 = 1-(1-f)^2$, the square of first eccentricity, and:

$$s = \sqrt{p_x^2 + p_y^2}.$$

After the initial guesses are calculated, the reduced latitude $\left(\bar{\beta}\right)$ is recalculated using

$$\beta = \mathrm{atan}\left(\frac{(1-f)\sin\mu}{\cos\mu}\right)$$

and geodetic latitude ($\bar{\mu}$) is reevaluated. This last step is repeated until $\bar{\mu}$ converges.

The altitude $\left(\bar{h}\right)$ above the planetary ellipsoid is calculated with

$$h = s\cos\mu + \left(p_z + e^2 N\sin\mu\right)\sin\mu - N,$$

where the radius of curvature in the vertical prime $(\bar{N})$ is given by

$$N = \frac{R}{\sqrt{1 - e^2(\sin\mu)^2}} \, .$$

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

[2] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. Second Edition. Reston, VA: AIAA Education Series, 2000.

[3] *Recommended Practice for Atmospheric and Space Flight Vehicle Coordinate Systems,* R-004-1992, ANSI/AIAA, February 1992.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

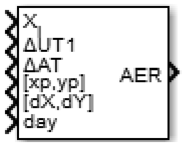
See Also

Direction Cosine Matrix ECEF to NED | Direction Cosine Matrix ECEF to NED to Latitude and Longitude | Geocentric to Geodetic Latitude | LLA to ECEF Position | Radius at Geocentric Latitude

Topics
“About Aerospace Coordinate Systems” on page 2-8

ECI Position to AER

Convert Earth-centered inertial (ECI) coordinates to azimuth coordinates



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The ECI Position to AER block converts Earth-centered inertial (ECI) position coordinates to azimuth, elevation, and slant-range coordinates (AER), based on the geodetic position (latitude, longitude, and altitude).

- Azimuth (A) — Angle measured clockwise from true north. It ranges from 0 to 360 degrees.
- Elevation (E) — Angle between a plane perpendicular to the ellipsoid and the line that goes from the local reference to the object position. It ranges from –90 to 90 degrees.
- Slant range (R) — Straight line distance between the local reference and the object.

Ports

Input

X_i — Position
3-by-1 element vector

Position, specified as a 3-by-1 element vector, in ECI coordinates.

Data Types: `double`

ΔUT1 — Difference between UTC and Universal Time
scalar

Difference between UTC and Universal Time (UT1) in seconds, specified as a scalar, for which the block calculates the direction cosine or transformation matrix.

Example: `0.234`

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

ΔAT — Difference between International Atomic Time and UTC
scalar

Difference between International Atomic Time (IAT) and UTC, specified as a scalar, in seconds, for which the function calculates the direction cosine or transformation matrix.

Example: 32

Dependencies

This port is disabled if the **Higher accuracy parameters** check box is cleared.

Data Types: `double`

[xp,yp] — Polar displacement of Earth
1-by-2 array

Polar displacement of Earth, specified as a 1-by-2 array, in radians, from the motion of the Earth crust, along the *x*-axis and *y*-axis.

Example: `[-0.0682e-5 0.1616e-5]`

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

Port_5 — Adjustment based on reduction method
1-by-2 array

Adjustment based on reduction method, specified as 1-by-2 array. The name of the port depends on the setting of the **Reduction** parameter:

- If the reduction method is `IAU-2000/2006`, this input is the adjustment to the location of the Celestial Intermediate Pole (CIP), specified in radians. This location ([d*X*,d*Y*]) is along the *x*-axis and *y*-axis, for example, `[-0.2530e-6 -0.0188e-6]`.
- If the reduction method is `IAU-76/FK5`, this input is the adjustment to the longitude ([$\Delta\delta\psi$, $\Delta\delta\varepsilon$]), specified in radians.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/Products page.

Example: `[-0.2530e-6 -0.0188e-6]`

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

Port_6 — Time increment source
scalar

Time increment source, specified as a scalar, such as the Clock block.

Dependencies

- The port name and time increment depend on the **Time Increment** parameter.

Time Increment Value	Port Name
`Day`	**day**
`Hour`	**hour**
`Min`	**min**

Time Increment Value	Port Name
Sec	**sec**
None	No port

- To disable this port, set the **Time Increment** parameter to None.

Data Types: double

Output

AER — Azimuth, elevation, and slant range
3-by-1 element vector

Local reference coordinates azimuth (degrees), elevation (degrees), and slant range (meters), specified as a 3-by-1 element vector.

Data Types: double

Parameters

Reduction — Reduction method

IAU-76/FK5 (default) | IAU-2000/2006

Reduction method to convert the coordinates. Method can be one of:

- IAU-76/FK5

  Reduce the calculation using the International Astronomical Union 76/Fifth Fundamental Catalogue (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, the block calculates the transformation matrix rather than the direction cosine matrix.

- IAU-2000/2006

  Reduce the calculation using the International Astronomical Union 2000/2006 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation.

Programmatic Use
Block Parameter: red
Type: character vector
Values: 'IAU-2000/2006' | 'IAU-76/FK5'
Default: 'IAU-2000/2006'

Year — Year

2014 (default) | double, whole number, greater than 1

Year to calculate the Coordinated Universal Time (UTC) date. Enter a double value that is a whole number greater than 1, such as `2014`.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: double, whole number, greater than 1
Default: `'2013'`

Month — Month

`January` (default) | `February` | `March` | `April` | `May` | `June` | `July` | `August` | `September` | `October` | `November` | `December`

Month to calculate the UTC date.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Day

`1` (default) | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10` | `11` | `12` | `13` | `14` | `15` | `16` | `17` | `18` | `19` | `20` | `21` | `22` | `23` | `24` | `25` | `26` | `27` | `28` | `29` | `30` | `31`

Day to calculate the UTC date.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'` | `'10'` | `'11'` | `'12'` | `'13'` | `'14'` | `'15'` | `'16'` | `'17'` | `'18'` | `'19'` | `'20'` | `'21'` | `'22'` | `'23'` | `'24'` | `'25'` | `'26'` | `'27'` | `'28'` | `'29'` | `'30'` | `'31'`
Default: `'1'`

Hour — Hour

`0` (default) | double, whole number, `0` to `24`

Hour to calculate the UTC date. Enter a double value that is a whole number, from `0` to `24`.

Programmatic Use
Block Parameter: `hour`
Type: character vector
Values: double, whole number, `0` to `24`
Default: `'0'`

Minutes — Minutes

`0` (default) | double, whole number, `0` to `60`

Minutes to calculate the UTC date. Enter a double value that is a whole number, from `0` to `60`.

Programmatic Use
Block Parameter: `min`
Type: character vector
Values: double, whole number, `0` to `60`
Default: `'0'`

Seconds — Seconds

`0` (default)

Seconds to calculate the UTC date. Enter a double value that is a whole number, from `0` to `60`.

Programmatic Use
Block Parameter: `sec`
Type: character vector
Values: double, whole number, `0` to `60`
Default: `'0'`

Time increment — Time increment

`None` (default) | `Day` | `Hour` | `Min` | `Sec`

Time increment between the specified date and the desired model simulation time. The block adjusts the calculated direction cosine matrix to take into account the time increment from model simulation. For example, selecting `Day` and connecting a simulation timer to the port means that each time increment unit is one day and the block adjusts its calculation based on that simulation time.

This parameter corresponds to the time increment input, the clock source.

If you select `None`, the calculated Julian date does not take into account the model simulation time.

Programmatic Use
Block Parameter: `deltaT`
Type: character vector
Values: `'None'` | `'Day'` | `'Hour'` | `'Min'` | `'Sec'`
Default: `'Day'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as follows.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `errorflag`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`

Higher accuracy parameters — Enable higher accuracy parameters

`on` (default) | `off`

Select this check box to allow the following as block inputs. These inputs let you better control the conversion result. See “Input” on page 5-397 for a description.

- Δ *UT1*
- Δ *AT*
- [*xp* , *yp*]
- [ΔδΨ, Δδε] or [d *X* ,d *Y*]

Programmatic Use
Block Parameter: `extraparamflag`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Units — Units

`Metric (MKS)` (default) | `English`

Specifies the parameter and output units.

Units	Position	Equatorial Radius	Altitude
`Metric (MKS)`	Meters	Meters	Meters
`English`	Feet	Feet	Feet

Dependencies

To enable this option, set **Earth model** to `WGS84`.

Programmatic Use
Block Parameter: `eunits`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Earth model — Earth model

`Custom` (default) | `WGS84`

Earth model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `earthmodel`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Earth model** to `Custom`.

Programmatic Use
Block Parameter: `flat`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius — Radius of planet at equator

`6378137` (default) | double scalar

Radius of the planet at its equator.

Dependencies

To enable this parameter, set **Earth model** to `Custom`.

Programmatic Use
Block Parameter: `eqradius`
Type: character vector
Values: double scalar
Default: `6378137`

Initial geodetic latitude and longitude [deg] — Initial geodetic latitude and longitude

`[0 0]` (default) | 2-by-1 vector

Reference location in latitude and longitude, specified as 2-by-1 vector, in degrees.

Programmatic Use
Block Parameter: `latlon0`
Type: character vector
Values: 2-by-1 vector
Default: `[0 0]`

Angular direction of the local reference system (degrees clockwise from north) — Angular direction

`0` (default) | scalar

Specifies angle for converting the flat Earth *x* and *y* coordinates to north and east coordinates, respectively. An example is the angle between the vessel and the true geodetic north.

Programmatic Use
Block Parameter: `psi0`
Type: character vector
Values: double scalar
Default: `0`

Reference height — Reference height

`0` (default) | scalar

Specifies the reference height measured from the surface of the Earth to the flat Earth frame. It uses the same units as the ECI position. Estimate the reference height relative to the Earth frame.

Programmatic Use
Block Parameter: `href`
Type: character vector
Values: double scalar
Default: `0`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

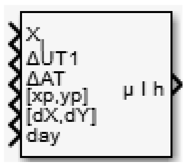
Generate C and C++ code using Simulink® Coder™.

See Also

LLA to ECI Position | ECI Position to LLA | Direction Cosine Matrix ECI to ECEF

ECI Position to LLA

Convert Earth-centered inertial (ECI) coordinates to geodetic latitude, longitude, altitude (LLA) coordinates



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The ECI Position to LLA block converts Earth-centered inertial (ECI) position coordinates to geodetic latitude, longitude, altitude (LLA) coordinates, based on the specified reduction method and Coordinated Universal Time (UTC), for the specified time and geophysical data.

Ports

Input

X_i — Original position
3-by-1 element vector

Original position vector with respect to the ECI reference system, specified as a 3-by-1 element vector.

Data Types: `double`

ΔUT1 — Difference between UTC and Universal Time
scalar

Difference between UTC and Universal Time (UT1) in seconds, specified as a scalar, for which the block calculates the direction cosine or transformation matrix.

Example: `0.234`

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

ΔAT — Difference between International Atomic Time and UTC
scalar

Difference between International Atomic Time (IAT) and UTC, specified as a scalar, in seconds, for which the block calculates the direction cosine or transformation matrix.

Example: `32`

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

[xp,yp] — Polar displacement of Earth
1-by-2 array

Polar displacement of Earth, specified as a 1-by-2 array, in radians, from the motion of the Earth crust, along the *x*-axis and *y*-axis.

Example: `[-0.0682e-5 0.1616e-5]`

Dependencies

To enable this port, select the **Higher accuracy parameters** check box.

Data Types: `double`

Port_5 — Adjustment based on reduction method
1-by-2 array

Adjustment based on reduction method, specified as 1-by-2 array. The name of the port depends on the setting of the **Reduction** parameter:

- If reduction method is `IAU-2000/2006`, this input is the adjustment to the location of the Celestial Intermediate Pole (CIP), specified in radians. This location ([d*X*,d*Y*]) is along the *x*-axis and *y*-axis, for example, `[-0.2530e-6 -0.0188e-6]`.
- If reduction method is `IAU-76/FK5`, this input is the adjustment to the longitude ([Δδψ, Δδε]), specified in radians.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/Products page.

Example: `[-0.2530e-6 -0.0188e-6]`

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

Port_6 — Time increment source
scalar

Time increment source, specified as a scalar, such as the Clock block.

Dependencies

- The port name and time increment depend on the **Time Increment** parameter.

Time Increment Value	Port Name
`Day`	**day**
`Hour`	**hour**
`Min`	**min**
`Sec`	**sec**
`None`	No port

- To disable this port, set the **Time Increment** parameter to `None`.

Data Types: `double`

Output

μ l h — Original position vector
3-by-1 element vector

Original position vector in geodetic LLA coordinates, returned as a 3-by-1 element vector, in degrees.

Data Types: `double`

Parameters

Reduction — Reduction method

`IAU-76/FK5` (default) | `IAU-2000/2006`

Reduction method to convert the coordinates. Method can be one of:

- `IAU-76/FK5`

  Reduce the calculation using the International Astronomical Union 76/Fifth Fundamental Catalogue (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, the block calculates the transformation matrix rather than the direction cosine matrix.

- `IAU-2000/2006`

  Reduce the calculation using the International Astronomical Union 2000/2006 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation.

Programmatic Use
Block Parameter: `red`
Type: character vector
Values: `'IAU-2000/2006'` | `'IAU-76/FK5'`
Default: `'IAU-2000/2006'`

Year — Year

`2014` (default) | double, whole number, greater than 1

Year to calculate the Coordinated Universal Time (UTC) date. Enter a double value that is a whole number greater than 1, such as `2014`.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: double, whole number, greater than 1

Default: '2013'

Month — Month

January (default) | February | March | April | May | June | July | August | September | October | November | December

Month to calculate the UTC date.

Programmatic Use
Block Parameter: month
Type: character vector
Values: 'January' | 'February' | 'March' | 'April' | 'May' | 'June' | 'July' | 'August' | 'September' | 'October' | 'November' | 'December'
Default: 'January'

Day — Day

1 (default) | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31

Day to calculate the UTC date.

Programmatic Use
Block Parameter: day
Type: character vector
Values: '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20' | '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31'
Default: '1'

Hour — Hour

0 (default) | double, whole number, 0 to 24

Hour to calculate the UTC date. Enter a double value that is a whole number, from 0 to 24.

Programmatic Use
Block Parameter: hour
Type: character vector
Values: double, whole number, 0 to 24
Default: '0'

Minutes — Minutes

0 (default) | double, whole number, 0 to 60

Minutes to calculate the UTC date. Enter a double value that is a whole number, from 0 to 60.

Programmatic Use
Block Parameter: min
Type: character vector
Values: double, whole number, 0 to 60
Default: '0'

Seconds — Seconds

0 (default)

Seconds to calculate the UTC date. Enter a double value that is a whole number, from 0 to 60.

Programmatic Use
Block Parameter: sec
Type: character vector
Values: double, whole number, 0 to 60
Default: '0'

Time increment — Time increment

None (default) | Day | Hour | Min | Sec

Time increment between the specified date and the desired model simulation time. The block adjusts the calculated direction cosine matrix to take into account the time increment from model simulation. For example, selecting Day and connecting a simulation timer to the port means that each time increment unit is one day and the block adjusts its calculation based on that simulation time.

This parameter corresponds to the time increment input, the clock source.

If you select None, the calculated Julian date does not take into account the model simulation time.

Programmatic Use
Block Parameter: deltaT
Type: character vector
Values: 'None' | 'Day' | 'Hour' | 'Min' | 'Sec'
Default: 'Day'

Action for out-of-range input — Out-of-range block behavior

None (default) | Warning | Error

Out-of-range block behavior, specified as follows.

Action	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: errorflag
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'Error'

Higher accuracy parameters — Enable higher accuracy parameters

on (default) | off

Select this check box to allow the following as block inputs. These inputs let you better control the conversion result. See “Input” on page 5-405 for a description.

- Δ *UT1*
- Δ *AT*
- [*xp* , *yp*]
- [Δδψ, Δδε] or [d *X* ,d *Y*]

Programmatic Use
Block Parameter: `extraparamflag`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Units — Output units

`Metric (MKS)` (default) | `English`

Specifies the parameter and output units.

Units	Position	Equatorial Radius	Altitude
`Metric (MKS)`	Meters	Meters	Meters
`English`	Feet	Feet	Feet

Dependencies

To enable this parameter, set **Earth model** to `Earth (WGS84)`.

Programmatic Use
Block Parameter: `eunits`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Earth model — Earth model

`Custom` (default) | `WGS84`

Earth model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `earthmodel`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of the planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Earth model** to `Custom`.

Programmatic Use
Block Parameter: `flat`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius — Radius

`6378137` (default) | scalar

Radius of the planet at its equator.

Dependencies

To enable this parameter, set **Earth model** to `Custom`.

Programmatic Use
Block Parameter: `eqradius`
Type: character vector
Values: double scalar
Default: `6378137`

Version History

Introduced in R2014a

Extended Capabilities

C/C++ Code Generation

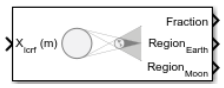
Generate C and C++ code using Simulink® Coder™.

See Also

LLA to ECI Position

Eclipse Shadow Model (Dual Cone)

Calculate fraction of solar disk visible at inertial positions



Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Alternative Configurations of Eclipse Shadow Model (Dual Cone) Block:
Eclipse Shadow Model (Cylindrical)

Description

The Eclipse Shadow Model (Dual Cone) block calculates the fraction of the solar disk that is visible from the provided inertial positions, assuming that the occulted bodies and the Sun are spherical. The block operates with two shadow models:

Dual cone model — The block differentiates between partial, annular, and total eclipse, which means the spacecraft can be in sunlight, penumbra, antumbra, or umbra. The eclipse is partial in penumbra, annular in antumbra, and total in umbra. In summary:

- Umbra, no sunlight — 0
- Penumbra or Antumbra, partial sunlight — 0 to 1
- Full sunlight — 1

Cylindrical model — The block differentiates only between total eclipse (umbra) and full sunlight. For more information, see “Alternative Configurations” on page 5-0 section.

- Umbra, no sunlight — 0
- Full sunlight — 1

Ports

Input

X_{icrf} — Position of spacecraft
3-element vector | *M*-by-3 array

Position of the spacecraft with respect to ICRF coordinate system, specified as a 3-element vector or *M*-by-3 array, where *M* is the number of spacecraft. The **Units** parameter controls the units designation.

Data Types: `double`

T_{JD} — Julian date
scalar

Julian date specified as a scalar.

Dependencies

To enable this port, set **Dialog** to `Port (Julian date)`.

Data Types: `double`

$T0_{JD}$ — Julian date epoch
scalar

Julian date epoch from which to base elapsed Julian time, specified as a scalar.

Dependencies

To enable this port, set **Dialog** to `Port (T0 and elapsed Julian time)`.

Data Types: `double`

ΔT_{JD} — Elapsed Julian time with respect to $T0_{JD}$
scalar

Elapsed Julian time with respect to **ΔT_{JD}**, specified as a scalar elapsed time.

Dependencies

To enable this port, set **Dialog** to `Port (Julian date)`.

Data Types: `double`

$R_{ss_{bary},cb}$ — Custom central body position
3-by-1 array

Custom central body position with respect to the solar system barycenter, in the ICRF coordinate frame.

Dependencies

To enable this port, set **Central body** to `Custom`.

Data Types: `double`

Output

Fraction — Fraction of visible Sun
scalar | *M*-element vector

Fraction of visible Sun (eclipse), returned as a scalar or vector of size *M*. *M* is the number of input spacecraft.

Dependencies

- When the **Shadow model** parameter is set to `Dual cone`, the values are `0` (Umbra), between `0` and `1` (Penumbra or Antumbra), or `1` (Sunlight).
- When the **Shadow model** parameter is set to `Cylindrical`, the values are `0` (Umbra) or `1` (Sunlight).

Data Types: `double`

Region — Region
scalar | *M*-element vector | integer

Region depending on **Shadow model** value, returned as a scalar or *M*-element vector of integers. *M* is the number of spacecraft.

- When the **Shadow model** parameter is set to `Dual cone`, region can be `0` (Umbra), between `0` and `1` (Penumbra or Antumbra), or `1` (Sunlight).
- When the **Shadow model** parameter is set to `Cylindrical`, region can be `0` (Umbra) or `1` (Sunlight).

Dependencies

To enable this port:

- Select the **Output index of shadow region** check box.

Data Types: `int32`

Secondary Region — Secondary region
scalar | *M*-element vector | integer

Region depending on **Shadow model** value, returned as a scalar or *M*-element vector of integers. *M* is the number of spacecraft.

- When the **Shadow model** parameter is set to `Dual cone`, region can be `0` (Umbra), between `0` and `1` (Penumbra or Antumbra), or `1` (Sunlight).
- When the **Shadow model** parameter is set to `Cylindrical`, region can be `0` (Umbra) or `1` (Sunlight).

Dependencies

To enable this port, select the **Output index of shadow region** check box and one of these settings:

- - Set **Central body** to `Earth`.
  - Select **Include Moon**.

or:

- - Set **Central body** to `Moon`.
  - Select **Include Earth**.

Data Types: `int32`

Parameters

Units — Position port units

`Metric (m)` (default) | `Metric (km)` | `English (ft)` | `English (M)`

Position port units, specified as one of these values:

Units	Distance
`Metric (m)`	meters
`Metric (km)`	kilometers
`English (ft)`	feet
`English (M)`	miles

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (m)'` | `'Metric (km)'` | `'English (ft)'` | `'English (M)'`
Default: `'Metric (m)'`

Shadow model — Shadow model

`Dual cone` (default) | `Cylindrical`

Shadow model for eclipse calculations, specified as one of these values.

- `Cylindrical` — Fraction can be 0.0 (Umbra) or 1.0 (Sunlight).
- `Dual cone` — Fraction can be 0.0 (Umbra), between 0.0 and 1.0 (Penumbra or Antumbra), or 1.0 (Sunlight).

Programmatic Use
Block Parameter: `shadowModel`
Type: character vector
Values: `'Dual cone'` | `'Cylindrical'`
Default: `'Dual cone'`

Output index of shadow region — Output index of shadow region

`on` (default) | `off`

To enable the port for the index of the shadow region, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `outputShadowRegion`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Time source — Time source method

`Dialog` (default) | `Port (Julian date)` | `Port (T0 and elapsed Julian time)`

Time source method, specified as one of these values:

- `Dialog` — Block dialog parameter.
- `Port (Julian date)` — $\mathbf{T}_{JD}$ port.
- `Port (T0 and elapsed Julian time)` — $\mathbf{T}_{JD}$ and $\mathbf{\Delta T}_{JD}$ ports.

Programmatic Use
Block Parameter: `timeSource`
Type: character vector
Values:`'Dialog'` | `'Port (Julian date)'` | `'Port (T0 and elapsed Julian time)'`
Default: `'Dialog'`

Start date/time (JD) — Initial start date and time for simulation

`juliandate(2020, 1, 1)` (default) | valid scalar Julian date

Initial start date and time of simulation, specified as a valid scalar Julian date. To determine the current time at each simulation timestep, the block adds elapsed simulation time to this value.

Tip To calculate the Julian date, use the `juliandate` function.

Dependencies

To enable this parameter, set **Time source** to `Dialog`.

Programmatic Use
Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1)'` | valid scalar Julian date
Default: `'juliandate(2020, 1, 1)'`

Central body — Occulting celestial body for eclipse calculations

`Earth` (default) | `Moon` | `Mercury` | `Venus` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Custom`

Occulting celestial body for eclipse calculations, specified as `Earth`, `Moon`, `Mercury`, `Venus`, `Mars`, `Jupiter`, `Saturn`, `Uranus`, `Neptune`, or `Custom`.

Position inputs are referenced with respect to the origin of the central body.

Programmatic Use
Block Parameter: `centralBody`
Type: character vector
Values: `'Earth'` | `'Moon'` |`'Mercury'` | `'Venus'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Custom'` |
Default: `'Earth'`

Custom radius — Custom central body radius

`3396200` (default) | double scalar

Equatorial radius for a custom central body, specified as a double scalar in the units specified in the **Units** parameter..

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `customR`
Type: character vector
Values: `'3396200'` | double scalar
Default: `'3396200'`

Include Moon — Option to include Moon

`on` (default) | `off`

Option to include Moon as a secondary occulting body in eclipse calculations when central body is Earth.

Dependencies

Set the **Central body** parameter to `Earth`.

Programmatic Use

Block Parameter: `includeMoon`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Include Earth — Option to include Earth

`on` (default) | `off`

Option to include Earth as a secondary occulting body in eclipse calculations when central body is Moon.

Dependencies

Set the **Central body** parameter to `Moon`.

Programmatic Use

Block Parameter: `includeEarth`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Ephemeris model — Ephemeris model

`DE405` (default) | `DE421` | `DE423` | `DE430` | `DE432t`

Select one of these ephemerides models defined by the Jet Propulsion Laboratory.

Ephemeris Model	Description
`DE405`	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
`DE421`	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
`DE423`	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Ephemeris Model	Description
DE430	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE432t	Released in April 2014. This ephemeris takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. In the MATLAB desktop toolstrip, click **Add-Ons**.

Programmatic Use
Block Parameter: `ephemerisModel`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Limit ephemerides date range — Open to enable start and end of range of ephemeris data

`on` (default) | `off`

Control how much data is loaded into memory during simulation and how much data is included in generated code for the block:

- Clear this check box to include data for the complete date range defined in the “Ephemeris model” on page 5-0 table.
- Select this check box to limit the loading of ephemeris data to a specified date range.

Programmatic Use
Block Parameter: `useDateRange`
Type: character vector
Values: `'off'` | `'on'` |
Default: `'on'`

Start date (JD) — Start date of ephemerides date range

`juliandate(2020, 1, 1)` (default) | Julian date

Start date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use
Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1)'` | Julian date
Default: `'juliandate(2020, 1, 1)'`

End date (JD) — End date of ephemerides date range

`juliandate(2050, 1, 1)` (default) | Julian date

End date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use
Block Parameter: `endDate`
Type: character vector
Values: `'juliandate(2050, 1, 1)'` | Julian format date
Default: `'juliandate(2050, 1, 1)'`

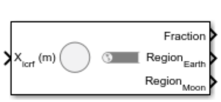
Action for out-of-range input — Out-of-range block behavior

`Error` (default) | `Warning` | `None`

Out-of-range block behavior, specified as one of these values:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`



Eclipse Shadow Model (Cylindrical) — Calculate fraction of solar disk visible at inertial positions

The Eclipse Shadow Model (Cylindrical) block calculates a fraction of the solar disk visible at inertial positions while assuming that:

- The Sun is infinitely far from the occulting bodies and the spacecraft.
- All rays of sunlight are parallel.

The block cannot differentiate between partial, annular, and total eclipse, which means the that the eclipse is either full sunlight or umbra only.

Differences between this block and the Eclipse Shadow Model (Cylindrical) include:

- **Shadow model** is `Cylindrical` by default.
- **Output index of shadow region** is cleared by default, resulting in only the output **Fraction** port appearing by default.

Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Version History

Introduced in R2023b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Orbit Propagator | Spacecraft Dynamics

Exhaust Gas Temperature (EGT) Indicator

Display measurements for engine exhaust gas temperature (EGT)

Libraries:
Aerospace Blockset / Flight Instruments

Description

The EGT Indicator block displays temperature measurements for engine exhaust gas temperature (EGT) in Celsius.

This block displays values using both:

- A needle on a gauge. A major tick is (**Maximum-Minimum**)/1,000 degrees, a minor tick is (**Maximum-Minimum**)/200 degrees Celsius.
- A numeric indicator. The operating range for the indicator goes from **Minimum** to **Maximum** degrees Celsius.

If the value of the signal is under **Minimum**, the needle displays 5 degrees under the **Minimum** value, the numeric display shows the **Minimum** value. If the value exceeds the **Maximum** value, the needle displays 5 degrees over the maximum tick, and the numeric displays the **Maximum** value.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name

Connect to signal for display, selected from list of signal names.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Minimum — Minimum tick mark value

0 (default) | finite | double | scalar

Minimum tick mark value, specified as a finite double, or scalar value, in ft/min.

Dependencies

The **Minimum** tick value must be less than the **Minimum** tick value.

Programmatic Use
Block Parameter: `Limits`
Type: double
Values: double scalar
Default: `[0 1000]`, where `0` is the minimum value

Maximum — Maximum tick mark value

1000 (default) | finite | double | scalar

Specify the maximum tick mark value, specified as a finite double, or scalar value, in ft/min..

Dependencies

The **Maximum** tick value must be greater than the **Maximum** tick value.

Programmatic Use
Block Parameter: `Limits`
Type: double
Values: double scalar
Default: `[0 1000]`, where `1000` is the maximum value

Scale Colors — Ranges of color bands
0 (default) | double | scalar

Ranges of color bands on the outside of the scale, specified as a finite double, or scalar value. Specify the minimum and maximum color range to display on the gauge.

To add a new color, click `+`. To remove a color, click `-`.

Programmatic Use
Block Parameter: `ScaleColors`
Type: *n*-by-1 struct array
Values: struct array with elements `Min`, `Max`, and `Color`

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is ignored for code generation.

See Also

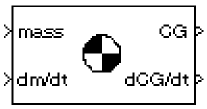
Airspeed Indicator | Altimeter | Artificial Horizon | Climb Rate Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Topics

"Display Measurements with Cockpit Instruments" on page 2-59
"Programmatically Interact with Gauge Band Colors" on page 2-61
"Flight Instrument Gauges" on page 2-58

Estimate Center of Gravity

Calculate center of gravity location



Libraries:
Aerospace Blockset / Mass Properties

Description

The Estimate Center of Gravity block calculates the center of gravity location and the rate of change of the center of gravity.

Linear interpolation is used to estimate the location of the center of gravity as a function of mass. The rate of change of the center of gravity is a linear function of the rate of change of mass.

Ports

Input

mass — Mass
scalar

Mass, specified as a scalar.

Data Types: `double`

dm/dt — Rate of change
scalar | 3-element vector

Rate of change of mass, specified as a scalar or three-element vector.

Data Types: | `double`

Output

CG — Center of gravity location
3-element vector

Center of gravity location, returned as a three-element vector.

Data Types: `double`

dCG/dt — Rate of change
3-element vector

Rate of the change of center of gravity location, returned as a three-element vector.

Data Types: `double`

Parameters

Full mass — Mass

2 (default) | scalar

Gross mass of the vehicle, specified as a double scalar.

Programmatic Use
Block Parameter: `fmass`
Type: character vector
Values: double scalar
Default: `'2'`

Empty mass — Empty mass

1 (default) | scalar

Empty mass of the vehicle, specified as double scalar.

Programmatic Use
Block Parameter: `emass`
Type: character vector
Values: double scalar
Default: `'1'`

Full center of gravity — Full center of gravity

`[1 1 1]'` (default) | 3-element vector

Center of gravity at the gross mass of the vehicle, specified as a three-element vector.

Programmatic Use
Block Parameter: `fcg`
Type: character vector
Values: 3-element vector
Default: `[1 1 1]'`

Empty center of gravity — Empty center of gravity

`[0.5 0.5 0.5]'` (default) | 3-element vector

Center of gravity at the empty mass of the vehicle, specified as a three-element vector.

Programmatic Use
Block Parameter: `ecg`
Type: character vector
Values: 3-element vector
Default: `[0.5 0.5 0.5]'`

Version History

Introduced before R2006a

Extended Capabilities

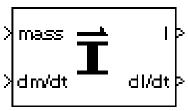
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Aerodynamic Forces and Moments | Estimate Inertia Tensor | Moments about CG due to Forces

Estimate Inertia Tensor

Calculate inertia tensor



Libraries:
Aerospace Blockset / Mass Properties

Description

The Estimate Inertia Tensor block calculates the inertia tensor and the rate of change of the inertia tensor.

Linear interpolation is used to estimate the inertia tensor as a function of mass. The rate of change of the inertia tensor is a linear function of rate of change of mass.

Ports

Input

mass — Mass
scalar

Mass, specified as a scalar.

Data Types: `double`

dm/dt — Rate of change
scalar

Rate of change of mass, specified as a scalar.

Data Types: `double`

Output

I — Inertia tensor
3-by-3 matrix

Inertia tensor, returned as a 3-by-3 matrix.

Data Types: `double`

dI/dt — Rate of change
3-by-3 matrix

Rate of change of inertia tensor, returned as a 3-by-3 matrix.

Data Types: `double`

Parameters

Full mass — Mass

2 (default) | scalar

Gross mass of the vehicle, specified as a double scalar.

Programmatic Use
Block Parameter: `fmass`
Type: character vector
Values: double scalar
Default: `'2'`

Empty mass — Empty mass

1 (default) | scalar

Empty mass of the vehicle, specified as a double scalar.

Programmatic Use
Block Parameter: `emass`
Type: character vector
Values: double scalar
Default: `'1'`

Full inertia matrix — Full inertia matrix

`eye(3)` (default) | 3-element matrix

Inertia tensor at gross mass of the vehicle, specified as a three-element matrix.

Programmatic Use
Block Parameter: `fI`
Type: character vector
Values: double scalar
Default: `'eye(3)'`

Empty inertia matrix — Empty inertia matrix

`eye(3)/2` (default) | scalar

Inertia tensor at empty mass of the vehicle, specified as a scalar.

Programmatic Use
Block Parameter: `eI`
Type: character vector
Values: scalar
Default: `'eye(3)/2'`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Estimate Center of Gravity | Symmetric Inertia Tensor

Rodrigues to Direction Cosine Matrix

Convert Euler-Rodrigues vector to direction cosine matrix

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Rodrigues to Direction Cosine Matrix block determines the 3-by-3 direction cosine matrix from a three-element Euler-Rodrigues vector. The Euler-Rodrigues vector input and resulting direction cosine matrix represent a right-hand passive transformation from frame A to frame B. The resulting direction cosine matrix represent a series of right-hand intrinsic passive rotations from frame A to frame B. For more information on Euler-Rodrigues vectors, see "Algorithms" on page 5-430.

Ports

Input

rod — Euler-Rodrigues vector
three-element vector

Euler-Rodrigues vector from which to determine the direction cosine matrix.

Data Types: `double`

Output

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix determined from the Euler-Rodrigues vector.

Data Types: `double`

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x \ b_y \ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

Version History

Introduced in R2017a

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rodrigues | Rodrigues to Quaternions | Rodrigues to Rotation Angles | Quaternions to Rodrigues | Rotation Angles to Rodrigues

Rodrigues to Quaternions

Convert Euler-Rodrigues vector to quaternion

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Rodrigues to Quaternions block determines the 4-by-1 quaternion from a three-element Euler-Rodrigues vector. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. The Euler-Rodrigues vector input and the resulting quaternion represent a right-hand passive transformation from frame A to frame B. The resulting quaternion represents a series of right-hand intrinsic passive rotations from frame A to frame B. For more information on Euler-Rodrigues vectors, see “Algorithms” on page 5-432.

Ports

Input

rod — Euler-Rodrigues vector
three-element vector

Euler-Rodrigues vector from which to determine the quaternion.

Data Types: `double`

Output

q — Quaternion
4-by-1 matrix

Quaternion determined from the Euler-Rodrigues vector.

Data Types: `double`

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x \ b_y \ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

Version History

Introduced in R2017a

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rodrigues | Rodrigues to Direction Cosine Matrix | Rodrigues to Rotation Angles | Quaternions to Rodrigues | Rotation Angles to Rodrigues

Rodrigues to Rotation Angles

Convert Euler-Rodrigues vector to rotation angles

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Rodrigues to Rotation Angles block converts the three-element Euler-Rodrigues vector into rotation angles. The Euler-Rodrigues angles represent a right-hand passive transformation from frame A to frame B. The resulting rotation angles represent a series of right-hand intrinsic passive rotations from frame A to frame B. For more information on Euler-Rodrigues vectors, see “Algorithms” on page 5-435.

Ports

Input

rod — Euler-Rodrigues vector
three-element vector

Euler-Rodrigues vector determined from rotation angles.

Data Types: `double`

Output

R1,R2,R3 — Rotation angles
three-element vector

Rotation angles, in radians, from which to determine the Euler-Rodrigues vector. Quaternion scalar is the first element.

Data Types: `double`

Parameters

Rotation order — Rotation order
ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation order for three wind rotation angles.

For the 'ZYX', 'ZXY', 'YXZ', 'YZX', 'XYZ', and 'XZY' rotations, the block generates an R2 angle that lies between ±pi/2 radians (±90 degrees), and R1 and R3 angles that lie between ±pi radians (±180 degrees).

For the 'ZYZ', 'ZXZ', 'YXY', 'YZY', 'XYX', and 'XZX' rotations, the block generates an R2 angle that lies between 0 and pi radians (180 degrees), and R1 and R3 angles that lie between ±pi (±180 degrees). However, in the latter case, when R2 is 0, R3 is set to 0 radians.

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: `'ZYX'` | `'ZYZ'` |`'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`
Default: `'ZYX'`

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x \ b_y \ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

Version History

Introduced in R2017a

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

Extended Capabilities

C/C++ Code Generation

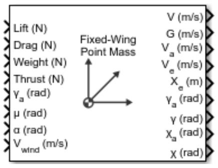
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rodrigues | Rodrigues to Direction Cosine Matrix | Rodrigues to Quaternions | Quaternions to Rodrigues | Rotation Angles to Rodrigues

Fixed-Wing Point Mass

Integrate fourth- or sixth-order point mass equations of motion in coordinated flight



Libraries:
Aerospace Blockset / Equations of Motion / Point Mass
UAV Toolbox / Algorithms

Description

The Fixed-Wing Point Mass block integrates fourth- or sixth-order point mass equations of motion in coordinated flight.

Limitations

- The flat Earth reference frame is considered inertial, an approximation that allows the forces due to the Earth's motion relative to the "fixed stars" to be neglected.
- The block assumes that there is fully coordinated flight, that is, there is no side force (wind axes) and sideslip is always zero.

Ports

Input

Lift — Lift
scalar

Lift, specified as a scalar in units of force.

Data Types: `double`

Drag — Drag
scalar

Drag, specified as a scalar in units of force.

Data Types: `double`

Weight — Weight
scalar

Weight, specified as a scalar in units of force.

Data Types: `double`

Thrust — Thrust
scalar

Thrust, specified as a scalar in units of force.

Data Types: `double`

γ_a — Flight path angle relative to the air mass
scalar

Flight path angle relative to the air mass, specified as a scalar in radians.

Data Types: `double`

μ — Bank angle
scalar

Bank angle, specified as a scalar in radians.

Data Types: `double`

α — Angle of attack
scalar

Angle of attack, specified as a scalar in radians.

Data Types: `double`

V_{wind} — Wind vector
three-element vector

Wind vector in the direction in which the air mass is moving, specified as a three-element vector.

Data Types: `double`

Output

V — Airspeed
scalar

Airspeed, returned as a scalar.

Data Types: `double`

G — Ground speed projection
scalar

Ground speed over the Earth (speed of motion over the ground), returned as a scalar.

Data Types: `double`

V_a — Velocity vector relative to air mass
three-element vector

Velocity vector relative to the air mass, returned as a three-element vector.

Data Types: `double`

V_e — Velocity vector relative to Earth with [North East Down] orientation
three-element vector

Velocity vector relative to Earth with `[North East Down]` orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [North East Down].

Data Types: double

V_{ENU} — Velocity vector relative to Earth
three-element vector

Velocity vector relative to Earth with [East North Up] orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [East North Up].

Data Types: double

X_e — Position vector relative to Earth
three-element vector

Position vector relative to Earth with [North East Down] orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [North East Down].

Data Types: double

X_{ENU} — Position vector relative to Earth
three-element vector

Position vector relative to Earth with [East North Up] orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [East North Up].

Data Types: double

γ_a — Flight path angle relative to air mass
scalar

Flight path angle relative to the air mass, returned as a scalar.

Data Types: double

γ — Flight path angle relative to Earth
scalar

Flight path angle relative to Earth, returned as a scalar.

Data Types: double

χ_a — Heading angle relative to air mass
scalar

Heading angle relative to air mass, returned as a scalar.

Dependencies

To enable this port, set **Degrees of Freedom** to `6th Order (Coordinated Flight)`.

Data Types: `double`

χ — Heading angle relative to Earth
scalar

Heading angle relative to Earth, returned as a scalar.

Dependencies

To enable this port, set **Degrees of Freedom** to `6th Order (Coordinated Flight)`.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (velocity in ft/s)` | `English (velocity in kts)`

Input and output units, specified as follows:

Units	Forces	Velocity	Position	Mass
`Metric (MKS)`	newtons	meters per second	meters	kilograms
`English (velocity in ft/s)`	pounds	feet per second	feet	slugs
`English (velocity in kts)`	pounds	knots	feet	slugs

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (velocity in ft/s)'` | `'English (velocity in kts)'`
Default: `'Metric (MKS)'`

Reference frame orientation — Reference frames

`[North East Down]` (default) | `[East North Up]`

Reference frames used for input ports and output ports, specified as `[East North Up]` or `[North East Down]`.

Programmatic Use
Block Parameter: `frame`
Type: character vector
Values: `'[East North Up]'` | `'[North East Down]'`
Default: `'[North East Down]'`

Degrees of freedom — Degrees of freedom

`6th Order (Coordinated Flight)` (default) | `4th Order (Longitudinal)`

Degrees of freedom, specified as `4th Order (Longitudinal)` or `6th Order (Coordinated Flight)`.

Programmatic Use
Block Parameter: `order`
Type: character vector
Values: `'4th Order (Longitudinal)'` | `'6th Order (Coordinated Flight)'`
Default: `'6th Order (Coordinated Flight)'`

Initial crossrange — Initial East (Earth) crossrange location

`0` (default) | scalar

Initial East (Earth) location in the `[North East Down]` orientation, specified as a scalar.

Dependencies

The direction specification of this parameter depends on the **Reference frame orientation** and **Degrees of Freedom** setting:

Initial crossrange	Reference frame orientation	Degrees of freedom
East	[North East Down]	6th Order (Coordinated Flight)
North	[East North Up]	6th Order (Coordinated Flight)

Programmatic Use
Block Parameter: `east`
Type: character vector
Values: scalar
Default: `'0'`

Initial downrange — Initial North (Earth) downrange

`0` (default) | scalar

Initial North (Earth) downrange of the point mass, specified as a scalar.

Dependencies

The direction specification of this parameter depends on the **Reference frame orientation** and **Degrees of Freedom** setting:

Initial downrange	Reference frame orientation	Degrees of freedom
North	[North East Down]	6th Order (Coordinated Flight)
North	[North East Down]	4th Order (Longitudinal)
East	[East North Up]	6th Order (Coordinated Flight)
East	[East North Up]	4th Order (Longitudinal)

Programmatic Use
Block Parameter: `north`
Type: character vector
Values: scalar

Default: `'0'`

Initial altitude — Initial altitude

`0` (default) | scalar

Initial altitude of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `altitude`
Type: character vector
Values: scalar
Default: `'0'`

Initial airspeed — Initial airspeed

`50` (default) | scalar

Initial airspeed of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `'airspeed'`
Type: character vector
Values: scalar
Default: `'50'`

Initial flight path angle — Initial flight path angle

`0` (default) | scalar

Initial flight path angle of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `gamma`
Type: character vector
Values: scalar
Default: `'0'`

Initial heading angle — Initial heading angle

`0` (default) | scalar

Initial heading angle of the point mass, specified as a scalar.

Dependencies

To enable this parameter, set **Degrees of Freedom** to `6th Order (Coordinated Flight)`.

Programmatic Use
Block Parameter: `chi`
Type: character vector
Values: scalar
Default: `'0'`

Mass — Point mass

`10` (default) | scalar

Mass of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: scalar
Default: `'10'`

Algorithms

The integrated equations of motion for the point mass are:

$$\dot{V} = (T\cos\alpha - D - W\sin\gamma_{ai})/m$$
$$\dot{\gamma}_a = ((L + T\sin\alpha)\cos\mu - W\cos\gamma_{ai})/(mV)$$
$$\dot{X}_e = V_a + V_w$$

6th order equations:

$$\dot{X}_a = ((L + T\sin\alpha)\sin\mu)/(mV\cos\gamma_a)$$
$$\dot{X}_a\Big|_{East} = V\cos\chi_a\cos\gamma_a$$
$$\dot{X}_a\Big|_{North} = V\sin\chi_a\cos\gamma_a$$
$$\dot{X}_a\Big|_{Up} = V\sin\gamma_a$$

4th order equations:

$$\dot{\chi}_a = 0$$
$$\dot{X}_a\Big|_{East} = V\cos\gamma_a$$
$$\dot{X}_a\Big|_{North} = 0$$
$$\dot{X}_a\Big|_{Up} = V\sin\gamma_a$$

where:

- m — Mass.
- g — Gravitational acceleration.
- W — Weight ($m*g$).
- L — Lift force.
- D — Drag force.
- T — Thrust force.
- α — Angle of attack.
- μ — Angle of bank.
- γ_{ai} — Input port value for the flight path angle.
- V — Airspeed, as measured on the aircraft, with respect to the air mass. It is also the magnitude of vector V_a.

- V_w — Steady wind vector.
- Subscript a — For the variables, denotes that they are with respect to the steadily moving air mass:
  - γ_a — Flight path angle.
  - χ_a — Heading angle.
  - X_a — Position [East, North, Up].
- Subscript e — Flat Earth inertial frame such that so X_e is the position on the Earth after correcting X_a for the air mass movement.

Additional outputs are:

$$G = \sqrt{(V_e|_{East}{}^2 + V_e|_{North}{}^2)}$$

$$\gamma = \sin^{-1}\left(\frac{V_e|_{Up}}{\|\underline{V_e}\|}\right)$$

$$\chi = \tan^{-1}\left(\frac{V_e|_{North}}{V_e|_{East}}\right)$$

where:

- The four-quadrant inverse tangent (`atan2`) calculates the heading angle.
- The groundspeed, G, is the speed over the flat Earth (a 2-D projection).

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

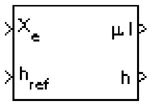
Generate C and C++ code using Simulink® Coder™.

See Also

4th Order Point Mass (Longitudinal) | 4th Order Point Mass Forces (Longitudinal) | 6th Order Point Mass (Coordinated Flight) | 6th Order Point Mass Forces (Coordinated Flight) | 6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Flat Earth to LLA

Estimate geodetic latitude, longitude, and altitude from flat Earth position



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Flat Earth to LLA block converts a 3-by-1 vector of flat Earth position($\bar{p}$) into geodetic latitude ($\bar{\mu}$), longitude ($\bar{\iota}$), and altitude (h). For more information on the flat Earth coordinate system, see "Algorithms" on page 5-448.

Limitations

- This estimation method assumes the flight path and bank angle are zero.
- This estimation method assumes the flat Earth *z*-axis is normal to the Earth at the initial geodetic latitude and longitude only. This method has higher accuracy over small distances from the initial geodetic latitude and longitude, and nearer to the equator. The longitude will have higher accuracy when there are smaller the variations in latitude. Additionally, longitude is singular at the poles.

Ports

Input

$\mathbf{X_e}$ — Position in flat Earth frame
3-by-1 vector

Position in flat Earth frame, specified as a 3-by-1 vector.

Data Types: `double`

$\mathbf{h_{ref}}$ — Reference height
scalar

Reference height from surface of Earth to flat Earth frame with regard to Earth frame, specified as a scalar in the same units as the flat Earth position.

Data Types: `double`

$\mathbf{\mu_{ref}\ l_{ref}}$ — Reference location
2-by-1 vector

Reference location, specified as a 2-by-1 vector, in degrees of latitude and longitude, for the origin of the estimation and the origin of the flat Earth coordinate system. Use this port if you want to specify the reference location as a dynamic value.

Dependencies

This port is enabled if the **Input reference position and orientation** check box is selected.

Data Types: `double`

ψ_{ref} — Direction of flat Earth *x*-axis
scalar

Angle, specified as a scalar, for converting flat Earth *x* and *y* coordinates to North and East coordinates. Use this port if you want to specify the angle as a dynamic value.

Dependencies

This port is enabled if the **Input reference position and orientation** check box is selected.

Data Types: `double`

Output

μ l — Geodetic latitude and longitude
2-by-1 vector

Geodetic latitude and longitude, returned as a 2-by-1 vector, in degrees.

Data Types: `double`

h — Altitude
scalar

Altitude above the input reference altitude, returned as a scalar, in the same units as the flat Earth position.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Parameter and output units.

Units	Position	Equatorial Radius	Altitude
`Metric (MKS)`	Meters	Meters	Meters
`English`	Feet	Feet	Feet

Dependencies

To enable this parameter, set **Planet model** to `Earth (WGS84)`.

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, specified as either `Custom` or `Earth (WGS84)`.

Dependencies

Selecting the `Custom` option enables these parameters:

- **Flattening**
- **Equatorial radius of planet**

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius of planet — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator, specified as a double scalar, in the same units as the **Units** parameter.

Dependencies

This parameter is enabled when **Planet model** is set to `Custom`.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `6378137`

Input reference position and orientation — Input reference position and orientation as ports

`off` (default) | `on`

Select this check box to enable ports for reference position and angle to convert flat Earth. Select this check box if you want to specify the reference positions and angle as dynamic values.

Dependencies

Selecting this check box replaces these parameters:

- **Reference geodetic latitude and longitude [deg]**
- **Direction of flat Earth x-axis (degrees clockwise from north)**

with these input ports:

- μ_{ref} l_{ref}
- ψ_{ref} input ports.

Programmatic Use

Block Parameter: `refPosPort`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Reference geodetic latitude and longitude [deg] — Initial geodetic latitude and longitude

`[0 10]` (default) | 2-by-1 vector

Reference location in latitude and longitude, specified as 2-by-1 vector, in degrees.

Dependencies

To enable this parameter, clear the **Input reference position and orientation** check box.

Programmatic Use

Block Parameter: `LL0`
Type: character vector
Values: 2-by-1 vector
Default: `[0 10]`

Direction of flat Earth x-axis (degrees clockwise from north) — Angle

`0` (default) | scalar

Angle to convert flat Earth x and y coordinates to North and East coordinates, specified as a scalar double, in degrees.

Dependencies

This parameter is disabled if the **Input reference position and orientation** check box is selected.

Programmatic Use

Block Parameter: `psi`
Type: character vector
Values: double scalar
Default: `0`

Algorithms

The flat Earth coordinate system assumes the *z*-axis is downward positive. The estimation begins by transforming the flat Earth *x* and *y* coordinates to North and East coordinates. The transformation has the form of:

$$\begin{bmatrix} N \\ E \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix},$$

where($\overline{\psi}$) is the angle in degrees clockwise between the *x*-axis and north.

To convert the North and East coordinates to geodetic latitude and longitude, the radius of curvature in the prime vertical (R_N) and the radius of curvature in the meridian (R_M) are used.

(R_N) and (R_M) are defined by the following relationships:

$$R_N = \frac{R}{\sqrt{1 - (2f - f^2)\sin^2\mu_0}}$$

$$R_M = R_N \frac{1 - (2f - f^2)}{1 - (2f - f^2)\sin^2\mu_0}$$

where (R) is the equatorial radius of the planet and$\left(\overline{f}\right)$ is the flattening of the planet.

Small changes in the latitude and longitude are approximated from small changes in the North and East positions by:

$$d\mu = \frac{dN}{R_M}$$

$$d\iota = \frac{dE}{(R_N \cos\mu_0)}.$$

The output latitude and longitude are simply the initial latitude and longitude plus the small changes in latitude and longitude:

$$\mu = \mu_0 + d\mu$$

$$\iota = \iota_0 + d\iota$$

The altitude is the negative flat Earth *z*-axis value minus the reference height (h_{ref}):

$$h = -p_z - h_{ref}.$$

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 2003.

[2] Etkin, B. *Dynamics of Atmospheric Flight* Hoboken, NJ: John Wiley & Sons, 1972.

Extended Capabilities

C/C++ Code Generation

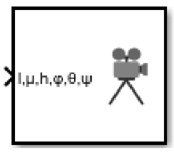
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix ECEF to NED | Direction Cosine Matrix ECEF to NED to Latitude and Longitude | ECEF Position to LLA | Geocentric to Geodetic Latitude | LLA to ECEF Position | Radius at Geocentric Latitude

FlightGear Preconfigured 6DoF Animation

Connect model to FlightGear flight simulator



Libraries:
Aerospace Blockset / Animation / Flight Simulator Interfaces

Description

The FlightGear Preconfigured 6DoF Animation block lets you drive position and attitude values to a FlightGear flight simulator vehicle given double-precision values for longitude (l), latitude (μ), altitude (h), roll (ϕ), pitch (θ), and yaw (ψ), respectively.

The block is configured as a sim viewing device. If you generate code for your model using Simulink Coder and connect to the running target code using external mode simulation, Simulink software can obtain the data from the target on the fly and transmit position and attitude data to FlightGear. For more information, see "Use C/C++ S-Functions as Sim Viewing Devices in External Mode".

The Aerospace Blockset product supports FlightGear versions starting from v2.6. If you are using a FlightGear version older than 2.6, the model displays a notification from the Simulink Upgrade Advisor. Consider using the Upgrade Advisor to upgrade your FlightGear version. For more information, see "Supported FlightGear Versions" on page 2-20.

Ports

Input

l,μ,h,ϕ,θ,ψ — Longitude, latitude, altitude, roll, pitch, and yaw
vector

Longitude, latitude, altitude, roll, pitch, and yaw, in double-precision, specified as a vector. Units are degrees west/north for longitude and latitude, meters above mean sea level for altitude, and radians for attitude values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Destination IP address — Destination IP address

`127.0.0.1` (default) | scalar

Destination IP address of the machine running FlightGear software, specified as a scalar.

Programmatic Use
Block Parameter: `DestinationIpAddress`

Type: character vector
Values: scalar
Default: `'127.0.0.1'`

Destination port — Destination port

scalar

Destination port of the machine running FlightGear software, specified as a scalar.

Programmatic Use
Block Parameter: `DestinationPort`
Type: character vector
Values: scalar
Default: `'5502'`

Sample time — Sample time

1/30 (default) | scalar

Sample time specified as a scalar (–1 for inherited).

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'1/30'`

Algorithms

The block is a masked subsystem containing principally a Pack net_fdm Packet for FlightGear block set for 6DoF inputs, a Send net_fdm Packet to FlightGear block, and a Simulation Pace block. To access the full capabilities of these blocks, use the individual corresponding blocks from the Aerospace Blockset library.

Version History

Introduced before R2006a

References

[1] Bowditch, N., *American Practical Navigator, An Epitome of Navigation*. US Navy Hydrographic Office, 1802.

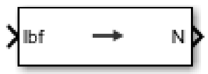
See Also

Generate Run Script | Pack net_fdm Packet for FlightGear | Receive net_ctrl Packet from FlightGear | Send net_fdm Packet to FlightGear | Unpack net_ctrl Packet from FlightGear

Topics
“Flight Simulator Interface” on page 2-20
“Work with the Flight Simulator Interface” on page 2-24

Force Conversion

Convert from force units to desired force units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Force Conversion block computes the conversion factor from specified input force units to specified output force units and applies the conversion factor to the input signal.

The Force Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Force
scalar | array

Force, specified as a scalar or array, in initial force units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Force
scalar | array

Force, returned as a scalar or array, in final force units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`lbf` (default) | `N`

Input units, specified as:

`lbf`	Pound force
`N`	Newtons

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `lbf` | `N`
Default: `lbf`

Final unit — Output units

`N` (default) | `lbf`

Output units, specified as:

`lbf`	Pound force
`N`	Newtons

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `lbf` | `N`
Default: `N`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

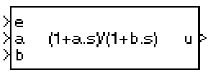
Generate C and C++ code using Simulink® Coder™.

See Also

Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

Gain Scheduled Lead-Lag

Implement first-order lead-lag with gain-scheduled coefficients



Libraries:
Aerospace Blockset / GNC / Control

Description

The Gain Scheduled Lead-Lag block implements a first-order lag of the form

$$u = \frac{1 + as}{1 + bs}e$$

where e is the filter input, and u is the filter output.

The coefficients a and b are inputs to the block. These values can depend on the flight condition or operating point. For example, you can produce them from the Lookup Table (n-D) Simulink block.

Ports

Input

e — Filter input
scalar

Filter input, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

a — Numerator coefficient
scalar

Numerator coefficient, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

b — Denominator coefficient
positive scalar

Denominator coefficient, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

u — Filter output
scalar

Filter output, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Initial state, x_initial — Initial internal state

0 (default) | vector

Initial internal state, specified as a vector, for the filter x_initial. Given this initial state, the initial output is given by

$$u|_{t=0} = \frac{x\_initial + ae}{b}$$

Programmatic Use
Block Parameter: initial state, x_initial
Type: character vector
Values: vector
Default: '0'

Version History
Introduced before R2006a

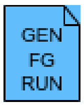
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Lookup Table (n-D)

Generate Run Script

Generate FlightGear run script on current platform



Libraries:
Aerospace Blockset / Animation / Flight Simulator Interfaces

Description

The Generate Run Script block generates a customized FlightGear run script on the current platform.

To generate the run script, fill in the required information in the Parameters fields, then click **Generate Script**.

In the dialog box, fields marked with an asterisk (*) are evaluated as MATLAB expressions. The other fields are treated as literal text.

Parameters

Select target architecture — Target platform to run script

`Default` (default) | `Win64` | `Linux` | `Mac`

From the list, select the target platform on which you want to execute the run script. This platform can differ from the platform on which you create the run script. Select `Default` if you want to generate a run script to run on the platform from which you create the run script.

- `Win64`
- `Linux`
- `Mac`

Programmatic Use
Block Parameter: `Architecture`
Type: character vector
Values: `'Win64' | 'Linux' | 'Mac'`
Default: `'Default'`

Select FlightGear data flow — FlightGear data flow

`Send` (default) | `Receive` | `Send-Receive`

From the list, select the direction of the data flow:

- `Send`

  Creates the run script to set up the sending of the `net_fdm` control model from Simulink to FlightGear.
- `Receive`

Creates the run script to set up the receiving of the `net_ctrl` control model from FlightGear to Simulink.

- `Send-Receive`

  Creates the run script to set up FlightGear to receive and broadcast data to and from Simulink.

  Note Selecting the `Send-Receive` option does not mean that you receive the same data that you sent (for example, you might not see control surface position data). With this option, you see primarily user input (such as data input via joystick) and environmental data.

Programmatic Use
Block Parameter: `dataFlow`
Type: character vector
Values: `'Receive' | 'Send-Receive'`
Default: `'Send'`

FlightGear geometry model name — Folder containing FlightGear geometry

HL20 (default)

Specify the name of the folder containing the model geometry that you want in the *`FlightGear`*`\data\Aircraft` folder.

Programmatic Use
Block Parameter: `GeometryModelName`
Type: character vector
Values:`'HL20'`
Default: `'HL20'`

Airport ID — ID of supported airport

KSFO (default)

ID of supported airport, selected from a list of supported airports available in the FlightGear interface, under **Location**.

Programmatic Use
Block Parameter: `'AirportId'`
Type: character vector
Values:`'KSFO'`
Default: `'KSFO'`

Runway ID — ID of supported runway

10L (default)

Specify the runway ID.

Programmatic Use
Block Parameter: `RunwayId`
Type: character vector
Values:`'10L'`
Default: `'10L'`

Initial altitude (ft)* — Initial aircraft altitude

7224 | numeric

Initial altitude of the aircraft, in feet.

Programmatic Use
Block Parameter: `InitialAltitude`
Type: character vector
Values: `'7224'`
Default: `'7224'`

Initial heading (deg)* — Initial aircraft heading

113 | numeric

Initial heading of the aircraft, in degrees.

Programmatic Use
Block Parameter: `InitialHeading`
Type: character vector
Values: `'113'`
Default: `'113'`

Offset distance (miles)* — Offset distance

4.72 | numeric

Offset distance of the aircraft from the airport, in miles.

Programmatic Use
Block Parameter: `OffsetDistance`
Type: character vector
Values: `'4.72'`
Default: `'4.72'`

Offset azimuth (deg)* — Aircraft offset azimuth

0 | numeric

Offset azimuth of the aircraft, in degrees.

Programmatic Use
Block Parameter: `OffsetAzimuth`
Type: character vector
Values: `'0'`
Default: `'0'`

Install FlightGear scenery during simulation (requires Internet connection) — Install FlightGear scenery

`off` (default) | `on`

Select this check box to direct FlightGear to automatically install required scenery while the simulator is running. Selecting this check box requires a stable Internet connection. For Windows systems, you may encounter an error message while launching FlightGear with this option enabled. For more information, see “Install Additional FlightGear Scenery” on page 2-22.

Programmatic Use
Block Parameter: `InstallScenery`
Type: character vector
Values:`'off' | 'on'`
Default: `'off'`

Disable FlightGear shader options — Disable FlightGear shader

`off` (default) | `on`

Select this check box to disable FlightGear shader options. Your computer built-in video card, such as NVIDIA cards, can conflict with FlightGear shaders. Consider selecting this check box if you have this conflict.

Programmatic Use
Block Parameter: `DisableShaders`
Type: character vector
Values:`'off' | 'on'`
Default: `'off'`

Destination/Origin IP address — Network IP address of machine running MATLAB

127.0.0.1

Network IP address of the machine on which MATLAB runs. This value is read-only.

Programmatic Use
Block Parameter: `OriginAddress`
Type: character vector
Values:`'127.0.0.1'`
Default: `'127.0.0.1'`

Destination port — Destination port of FlightGear machine

5502

Network flight dynamics model (fdm) port. For more information, see the Send net_fdm Packet to FlightGear block reference.

Programmatic Use
Block Parameter: `DestinationPort`
Type: character vector
Values:`'5502'`
Default: `'5502'`

Origin port — Origin port of FlightGear machine

5505

Network control (`ctrl`) port. For more information, see the Receive net_ctrl Packet from FlightGear block.

Programmatic Use
Block Parameter: `OriginPort`
Type: character vector
Values:`'5505'`
Default: `'5505'`

Network IP address — Network IP address of FlightGear machine

127.0.0.1

Network IP address of the machine on which the MATLAB software runs.

Programmatic Use
Block Parameter: `LocalAddress`
Type: character vector
Values:`'127.0.0.1'`
Default: `'127.0.0.1'`

Output file name — Output file

`runfg.bat`

Output file name. The file name is the name of the command that you use to start FlightGear with these initial parameters.

Note The run script file name must be composed of ASCII characters.

Use these file extensions:

Platform	Extension
Windows	`.bat`
Linux and macOS	`.sh`

Programmatic Use
Block Parameter: `OutputFileName`
Type: character vector
Values:`'runfg.bat'`
Default: `'runfg.bat'`

FlightGear base directory — FlightGear base directory

`C:\Program Files\FlightGear`

Specify the name of the FlightGear installation folder.

Note FlightGear must be installed in a folder path name composed of ASCII characters.

Programmatic Use
Block Parameter: `FlightGearBaseDirectory`
Type: character vector

Values:`'C:\Program Files\FlightGear'`
Default: `'C:\Program Files\FlightGear'`

Generate Script — Generate Script button

button

Click **Generate Script** to generate a run script for FlightGear. Do not click this button until you have entered the correct information in the dialog box parameters.

Version History

Introduced before R2006a

See Also

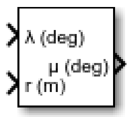
FlightGear Preconfigured 6DoF Animation | Pack net_fdm Packet for FlightGear | Receive net_ctrl Packet from FlightGear | Send net_fdm Packet to FlightGear | Unpack net_ctrl Packet from FlightGear

Topics

"Flight Simulator Interface" on page 2-20
"Work with the Flight Simulator Interface" on page 2-24

Geocentric to Geodetic Latitude

Convert geocentric latitude to geodetic latitude



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Geocentric to Geodetic Latitude block converts a geocentric latitude (λ) into geodetic latitude (μ) and optional ellipsoidal altitude (h) using geocentric latitude and the radius from the center of the planet to the center of gravity. The function uses an iteration-method of Bowring's formula to calculate the geodetic latitude. For more information, see “Algorithms” on page 5-464.

Limitations

This implementation generates a geodetic latitude that lies between ±90 degrees.

Ports

Input

λ — Geocentric latitude
scalar

Geocentric latitude, specified as a scalar, in degrees. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `double`

r — Radius
scalar

Radius from center of the planet to the center of gravity, specified as a scalar.

Data Types: `double`

Output

μ — Geodetic latitude
scalar

Geodetic latitude, specified as a scalar, in degrees.

Data Types: `double`

h — Ellipsoidal altitude
scalar

Ellipsoidal altitude, returned as a scalar.

Dependencies

To enable this port, select **Output altitude**.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Parameter and output units:

Units	Radius from CG to Center of Planet	Equatorial Radius
`Metric (MKS)`	Meters	Meters
`English`	Feet	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

This parameter is enabled when **Planet model** is set to `Custom`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius of planet — Radius

`6378137.0` (default) | scalar

Radius of the planet at its equator, in the same units as the **Units** parameter.

Dependencies

This parameter is enabled when **Planet model** is set to `Custom`.

Programmatic Use

Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `6378137`

Output altitude — Enable ellipsoidal altitude

`off` (default) | on

Select this check box to output the ellipsoidal altitude.

Dependencies

Select this check box to enable the **h** port.

Programmatic Use

Block Parameter: `outputAltitude`
Type: character vector
Values: `off` | `on`
Default: `'off'`

Algorithms

The Geocentric to Geodetic Latitude block converts a geocentric latitude (λ) into geodetic latitude (μ), where:

- λ — Geocentric latitude
- μ — Geodetic latitude
- r — Radius from the center of the planet
- f — Flattening
- a — Equatorial radius of the plant (semi-major axis)

Given geocentric latitude (λ) and the radius (r) from the center of the planet, this block first converts the desired points into the distance from the polar axis (ρ) and the distance from the equatorial axis (z).

$$\rho = r(\cos(\lambda))$$
$$z = r(\sin(\lambda)) .$$

It then calculates the geometric properties of the planet:

$$b = a(1 - f)$$
$$e^2 = f(2 - f)$$
$$e'^2 = \frac{e^2}{(1 - e^2)} .$$

And then uses the fixed-point iteration of Bowring's formula to calculate μ. This formula typically converges in three iterations.

$$\beta = \tan^{-1}\left(\frac{(1-f)\sin(\mu)}{\cos(\mu)}\right)$$

$$\mu = \tan^{-1}\left(\frac{z + be'^2\sin(\beta)^3}{\rho - ae^2\cos(\beta)^3}\right).$$

Version History

Introduced before R2006a

References

[1] Jackson, E. B., *Manual for a Workstation-based Generic Flight Simulation Program (LaRCsim) Version 1.4*, NASA TM 110164, April, 1995.

[2] Hedgley, D. R., Jr. "An Exact Transformation from Geocentric to Geodetic Coordinates for Nonzero Altitudes." NASA TR R-458, March, 1976.

[3] Clynch, J. R. "Radius of the Earth - Radii Used in Geodesy." Naval Postgraduate School, Monterey, California, 2002.

[4] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

[5] Edwards, C. H., and D. E. Penny. *Calculus and Analytical Geometry 2nd Edition*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

Extended Capabilities

C/C++ Code Generation

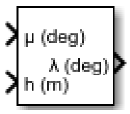
Generate C and C++ code using Simulink® Coder™.

See Also

ECEF Position to LLA | Flat Earth to LLA | Geodetic to Geocentric Latitude | LLA to ECEF Position

Geodetic to Geocentric Latitude

Convert geodetic latitude to geocentric latitude



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Geodetic to Geocentric Latitude block converts a geodetic latitude (μ) into geocentric latitude (λ) and optional radius from the center of the planet to the center of gravity (`r`) using geodetic latitude and ellipsoidal altitude. For more information on the geocentric latitude, see “Algorithms” on page 5-468.

Limitations

This block implementation generates a geocentric latitude that lies between ±90 degrees.

Ports

Input

μ — Geodetic latitude
scalar

Geodetic latitude, specified as a scalar, in degrees. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `double`

h — Ellipsoidal altitude
scalar

Ellipsoidal altitude, specified as a scalar.

Data Types: `double`

Output

λ — Geocentric latitude
scalar

Contains the geocentric latitude, specified as a scalar, in degrees.

Data Types: `double`

r — Radius
scalar

Radius from center of the planet to the center of gravity, returned as a scalar.

Dependencies

To enable this port, select **Output radius**.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Parameter and output units:

Units	Radius from CG to Center of Planet	Equatorial Radius
`Metric (MKS)`	Meters	Meters
`English`	Feet	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Dependencies

Selecting the `Custom` option enables these parameters:

- **Flattening**
- **Equatorial radius of planet**

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

This parameter is enabled when **Planet model** is set to `Custom`.

Programmatic Use

Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius of planet — Radius

`6378137.0` (default) | scalar

Radius of the planet at its equator, in the same units as the **Units** parameter.

Dependencies

This parameter is enabled when **Planet model** is set to `Custom`.

Programmatic Use

Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `6378137`

Output radius — Enable output of radius

`off` (default) | on

Select this check box to output the scalar distance radius from the equatorial radius to the center of the planet.

Dependencies

Select this check box to enable the **r** port.

Programmatic Use

Block Parameter: `outputRadius`
Type: character vector
Values: `off` | `on`
Default: `'off'`

Algorithms

The Geodetic to Geocentric Latitude block converts a geodetic latitude (μ) into geocentric latitude (λ), where:

- λ — Geocentric latitude
- μ — Geodetic latitude
- h — Height from the surface of the planet
- f — Flattening
- a — Equatorial radius of the plant (semi-major axis)

Given the geodetic latitude (μ) and the height from the surface of the planet (h), this block first calculates the geometric properties of the planet.

$$e^2 = f*(2 - f)$$
$$N = \frac{a}{\sqrt{1 - e^2\sin(\mu)^2)}}.$$

It then calculates the geocentric latitude from the point's distance from the polar axis (ρ) and distance from the equatorial axis (z).

$$\rho = (N + h)*\cos(\mu)$$
$$z = (N(1 - e^2) + h)\sin(\mu)$$
$$\lambda = \tan^{-1}\left(\frac{z}{\rho}\right).$$

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Topics
ECEF Position to LLA
Flat Earth to LLA
Geocentric to Geodetic Latitude
LLA to ECEF Position
Radius at Geocentric Latitude

Heading Indicator

Display measurements for aircraft heading

Libraries:
Aerospace Blockset / Flight Instruments

Description

The Heading Indicator block displays measurements for aircraft heading in degrees.

The block represents values between 0 and 360 degrees.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name

Connect to signal for display, selected from list of signal names.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is ignored for code generation.

See Also

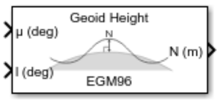
Airspeed Indicator | Altimeter | Artificial Horizon | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Revolutions Per Minute (RPM) Indicator | Turn Coordinator

Topics

"Display Measurements with Cockpit Instruments" on page 2-59
"Flight Instrument Gauges" on page 2-58

Geoid Height

Calculate undulations/height



Libraries:
Aerospace Blockset / Environment / Gravity

Description

The Geoid Height block calculates the geoid height using the **Geopotential model** parameter. The block interpolates the geoid heights from a grid of point values in the tide-free system. It uses the specified geopotential model to degree and order of the model. The geoid undulations are relative to the WGS84 ellipsoid.

The interpolation scheme wraps over the poles to allow for geoid height calculations at and near these locations.

Limitations

This block has the limitations of the selected geopotential model.

Ports

Input

μ (deg) — Geodetic latitude
scalar

Geodetic latitude, specified as a scalar, in degrees, where north latitude is positive and south latitude is negative. Input latitude must be of type single or double. If latitude is not in the range from –90 to 90, the block wraps it to be within the range.

Data Types: `double` | `single`

l (deg) — Longitude
scalar

Longitude, specified as a scalar, in degrees, where east longitude is positive in the range from 0 to 360. Input longitude must be of type single or double. If longitude is not in the range from 0 to 360, the block wraps it to be within the range when **Action for out-of-range input** is set to `None` or `Warning`. It does not wrap when **Action for out-of-range input** is set to `Error`.

Data Types: `double` | `single`

Output

N — geoid height
scalar

Geoid height, returned as a scalar, in selected length units. The data type is the same as the latitude in the first input.

Data Types: `double` | `single`

Parameters

Units — Parameter and output units

`Metric (MKS)` (default) | `English`

Parameter and output units, specified as:

Units	Height
`Metric (MKS)`	Meters
`English`	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Geopotential model — Geopotential model

`EGM96` (default) | `EGM2008` | `Custom`

Geopotential model, specified as:

Geopotential Model	Description
`EGM96 (Earth)`	Default. EGM96 Geopotential Model to degree and order 360. This model uses a 15-minute grid of point values in the tide-free system. This block calculates geoid heights to an accuracy of 0.01 m for this model.
`EGM2008 (Earth)`	EGM2008 Geopotential Model to degree and order 2159. This model uses a 2.5-minute grid of point values in the tide-free system. This block calculates geoid heights to an accuracy of 0.001 m for this model. **Note** This block requires that you download geoid data for the EGM2008 Geopotential Model with the Add-On Explorer. Click the **Get data** button to start the Add-On Explorer. For more information, see `aeroDataPackage`. If the data is installed, the **Get data** button does not appear.
`Custom`	Custom geopotential model that you define in **Geopotential mat-file**. This block calculates geoid heights to an accuracy of 0.01 m for custom models. Selecting `Custom` enables the **Geopotential mat-file** parameter.

Programmatic Use
Block Parameter: `gtype`
Type: character vector
Values: `'EGM96'` | `'EGM2008'` | `'Custom'`
Default: `'Earth'`

Geopotential mat-file — Geopotential MAT-file

`'geoidegm96grid'` (default) | MAT-file

Geopotential MAT-file that defines your custom geopotential model.

Dependencies

To enable this, set **Geopotential model** to `Custom`.

Programmatic Use
Block Parameter: `datafile`
Type: character vector
Values: `'geoidegm96grid'` | MAT-file
Default: `'geoidegm96grid'`

Data type — Data type of input and output signals

`double` (default) | `single`

Data type of the input and output signals, specified as `double` or `single`.

Programmatic Use
Block Parameter: `dtype`
Type: character vector
Values: `'double'` | `'single'`
Default: `'double'`

Action for out-of-range input — Out-of-range input behavior

`Warning` (default) | `Error` | `None`

Out-of-range input behavior (latitude outside –90 to 90 degrees, longitude outside 0 to 360 degrees), specified as follows.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer.
`Error` (default)	MATLAB returns an exception, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced in R2010b

References

[1] Vallado, David. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

[2] "Department of Defense World Geodetic System 1984, Its Definition, and Relationship with Local Geodetic Systems." NIMA TR8350.2.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

WGS84 Gravity Model | Spherical Harmonic Gravity Model

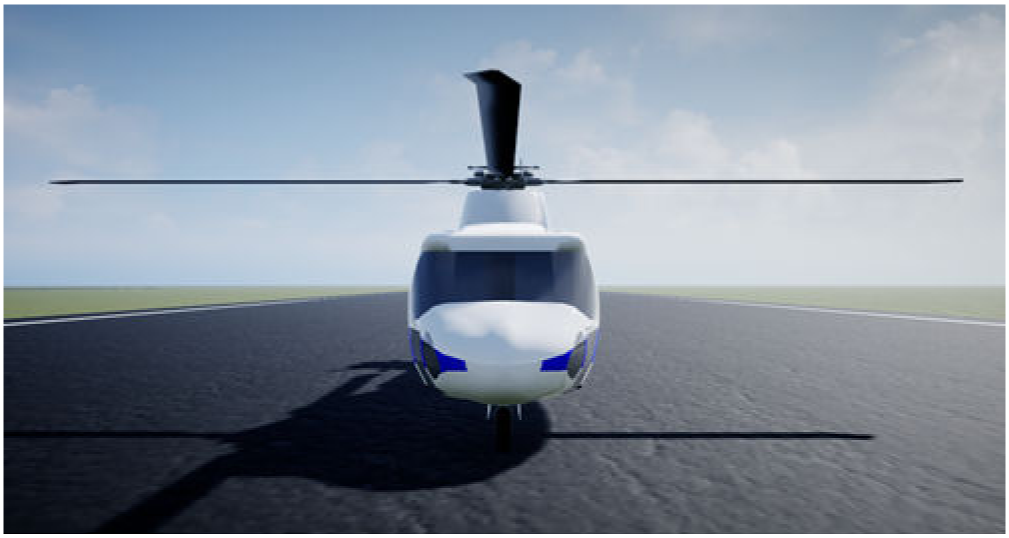
External Websites

Office of Geomatics

Helicopter

Generic helicopter

Description



Helicopter is one of the rotorcraft that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. For detailed views of the Helicopter, see “Views” on page 5-477.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Rotorcraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Parameters** tab, set the **Type** parameter to `Helicopter`.
3. Set the **Initial translation (m)** and **Initial rotation (rad)** parameters to an array size that matches the Helicopter rotorcraft, for example, `zeros(19, 3)`.

Data for Helicopter Placement

The `Helicopter` sample mesh origin is at ground level. To place the rotorcraft, consider using these values.

Airport Scene Placement

To place the `Light Helicopter` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, 0] + [0, 0, -0.01]`

Body Motion Ports and Parameters	Value
Rotation port and **Initial rotation** parameter	`[0, 0, 0]`

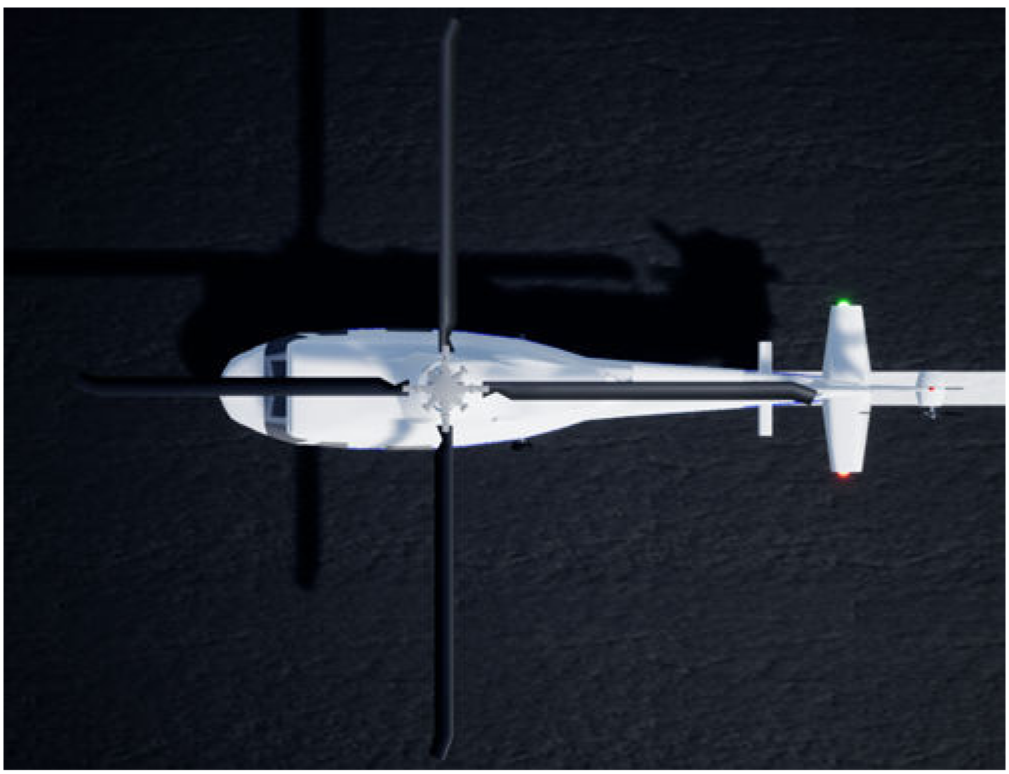
Altitude Sensor

For the altitude sensor in the Simulation 3D Rotorcraft block, use these values.

Parameter	Value
Body Z offset (m)	`1.1687`
Front tire radius (m)	`0.196`
Left tire radius (m)	`0.203`
Right tire radius (m)	`0.203`

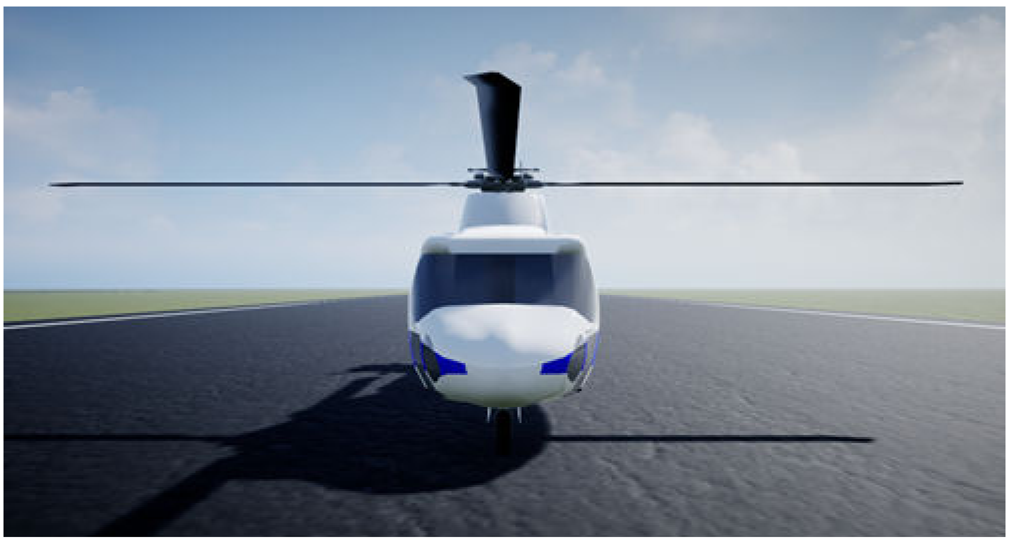
Views

`Top-down view` — Helicopter top-down view
diagram

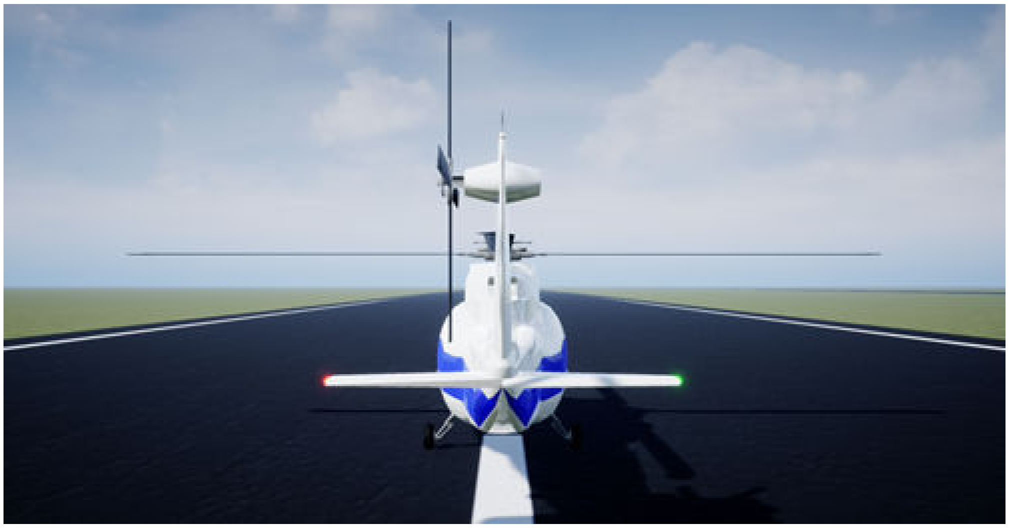


`Side view` — Helicopter side view
diagram

`Front view` — Helicopter front view
diagram



`Back view` — Helicopter back view
diagram



Lights and Skeleton

Lights

Light	Bone
Landing light	LandingLight
Nose light	NoseLight
Red navigation light	RedNavLight
Green navigation light	GreenNavLight
White navigation light	PositionLight
Strobe light	StrobeLight
Beacon light	BeaconLight

Skeleton

- `Helicopter`
  - `Engine1`
  - `Engine2`
  - `Rotor1`
  - `Rotor2`
  - `NoseGear`
    - `NoseGear_Wheel`
    - `NoseGear_Light`
  - `NoseGearDoor1`
  - `NoseGearDoor2`
  - `MainGear_L`

    - MainGear_L_Wheel
    - MainGear_L_Light
- MainGear_L_Door1
- MainGear_L_Door2
- MainGear_R
    - MainGear_R_Wheel
- MainGear_R_Door1
- MainGear_R_Door2
- Sensor1
- Sensor2
- RedNavLight
- GreenNavLight
- BeaconLight
- StrobeLight
- PositionLight

Version History

Introduced in R2023a

See Also

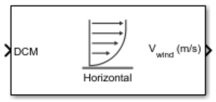
Simulation 3D Rotorcraft | Light Helicopter | Multirotor

Topics

“Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37
“Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2
“How 3D Simulation for Aerospace Blockset Works” on page 2-40

Horizontal Wind Model

Transform horizontal wind into body-axes coordinates



Libraries:
Aerospace Blockset / Environment / Wind

Description

The Horizontal Wind Model block computes the wind velocity in body-axes coordinates.

The wind is specified by wind speed and wind direction in Earth axes. The speed and direction can be constant or variable over time. The direction of the wind is in degrees clockwise from the direction of the Earth *x*-axis (north). The wind direction is defined as the direction from which the wind is coming. Using the direction cosine matrix (DCM), the wind velocities are transformed into body-axes coordinates.

Ports

Input

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix representing the flat Earth coordinates to body-fixed axis coordinates.

Data Types: `double`

V_{wind} — Wind speed
1-by-3 vector

Wind speed, specified as a 1-by-3 vector, in selected units.

Dependencies

To enable this parameter, set **Wind speed source** to `External`.

Data Types: `double`

θ_{wind} — Wind direction
scalar

Wind direction, specified as a scalar, in degrees. The direction of the wind is in degrees clockwise from the direction of the Earth *x*-axis (north). The wind direction is defined as the direction from which the wind is coming.

Dependencies

To enable this parameter, set **Wind direction source** to `External`.

Data Types: `double`

Output

V_{wind} — Wind velocity
3-by-3 matrix

Wind velocity, returned as a three-element signal in the same body coordinate reference as the **DCM** input, in specified units.

Data Types: `double`

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Wind Speed	Wind Velocity
`Metric (MKS)`	Meters per second	Meters per second
`English (Velocity in ft/s)`	Feet per second	Feet per second
`English (Velocity in kts)`	Knots	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in kts)'` | `'English (Velocity in ft/s)'`
Default: `'Metric (MKS)'`

Wind speed source — Wind speed source

`Internal` (default) | `External`

Wind speed source, specified as:

`External`	Variable wind speed input to block
`Internal`	Constant wind speed specified in mask

Dependencies

- Setting this parameter to `Internal` enables **Wind speed at altitude**.
- Setting this parameter to `External` enables the **V_{wind}** input port.

Programmatic Use
Block Parameter: `Vw_source`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Wind speed at altitude (m/s) — Wind speed

15 (default) | scalar

Constant wind speed, specified as a double scalar, in specified units.

Dependencies

To enable this parameter, set **Wind speed source** to `Internal`.

Programmatic Use
Block Parameter: `Vwind`
Type: character vector
Values: scalar
Default: `'15'`

Wind direction source — Wind direction source

`Internal` (default) | `External`

Wind direction source, specified as:

`External`	Variable wind direction input to block
`Internal`	Constant wind direction specified in mask

Dependencies

- Setting this parameter to `Internal` enables **Wind direction at altitude (degrees clockwise from north)**.
- Setting this parameter to `External` enables the $\boldsymbol{\theta}_{\mathbf{wind}}$ input port.

Programmatic Use
Block Parameter: `W_source`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Wind direction at altitude (degrees clockwise from north) — Wind direction

`0` (default)

Constant wind direction, specified as a scalar, in degrees clockwise from the direction of the Earth *x*-axis (north). The wind direction is the direction from which the wind is coming.

Dependencies

To enable this parameter, set **Wind direction source** to `Internal`.

Programmatic Use
Block Parameter: `Wdeg`
Type: character vector
Values: scalar
Default: `'0'`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Dryden Wind Turbulence Model (Continuous) | Dryden Wind Turbulence Model (Discrete) | Discrete Wind Gust Model | Horizontal Wind Model 07 | Horizontal Wind Model 14 | Von Karman Wind Turbulence Model (Continuous) | Wind Shear Model

Horizontal Wind Model 07

Implement Horizontal Wind Model 07

Libraries:
Aerospace Blockset / Environment / Wind

Description

The Horizontal Wind Model 07 block implements the U.S. Naval Research Laboratory HWM™ routine to calculate the meridional and zonal components of the wind for a set of geographic coordinates: latitude, longitude, and altitude.

Limitations

For code generation, use this block only for targets whose type is int 32 or higher.

Ports

Input

μ l h — Geodetic latitude, longitude, and geopotential altitude
three-element vector | altitude is a value between 0 and 500 km

Geodetic latitude (μ), longitude (l), and geopotential altitude (h), specified as a three-element vector.

Latitude and longitude values are in degrees.

Altitude values are held outside the range 0 to 500 km. The altitude value is in the units selected in **Units**.

Data Types: `double`

day — Day
scalar | value between 1 and 366

Day of year in Coordinated Universal Time (UTC), specified as a value between 1 and 366 (for a leap year). Values are wrapped within the range 1 to 366 days.

Data Types: `double`

sec — Elapsed seconds
scalar

Elapsed seconds since midnight for the specified day, in UTC.

Data Types: `double`

Ap — Ap index
scalar | range from 0 to 400

Ap index for the Universal Time (UT), specified as a scalar, ranging from 0 to 400. Select the index from NOAA National Geophysical Data Center, which contains 3 hour interval geomagnetic disturbance index values. If the Ap index value is greater than zero, the software takes into account magnetic effects during model evaluation.

Dependencies

To enable this port, set **Model** to `Total` or `Disturbance`.

Data Types: `double`

Output

V_{wind} — Wind velocity vector
1-by-2 vector

Wind velocity vector, returned as a 1-by-2 vector, containing the meridional and zonal wind components, in that order.

Data Types: `double`

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as:

Units	Wind Speed	Wind Velocity
`Metric (MKS)`	Meters per second	Meters per second
`English (Velocity in ft/s)`	Feet per second	Feet per second
`English (Velocity in kts)`	Knots	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Model — Horizontal wind model type

`Quiet` (default) | `Total` | `Disturbance`

Horizontal wind model type for which to calculate the wind components, specified as:

- `Disturbance`

  Calculate the effect of only magnetic disturbances in the wind.
- `Quiet`

  Calculate the horizontal wind model without magnetic disturbances.

- `Total`

  Calculate the combined effect of the quiet and magnetic disturbances.

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'Quiet'` | `'Total'` | `'Disturbance'`
Default: `'Quiet'`

Action for out-of-range input — Out-of-range block behavior

`Error` (default) | `Warning` | `None`

Out-of-range block behavior, specified as follows.

Value	Description
`None`	No action. The block imposes upper and lower limits on an input signal.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues. For Accelerator and Rapid Accelerator modes, setting the action to `Warning` has no effect and the model behaves as though the action is set to `None`.
`Error`	MATLAB returns an exception, model simulation stops. For Accelerator and Rapid Accelerator modes, setting the action to `Error` has no effect and the model behaves as though the action is set to `None`.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced in R2014b

R2021b: Horizontal Wind Model 07 Block Possible Changed Returned Values

Behavior changed in R2021b

The Horizontal Wind Model 07 block now accepts:

- **day** port values that are decimal, negative, `0`, or greater than `366`.
- **sec** port values that are `0` or greater than `86400`.

As a result, the output values from this block might change from previous releases.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Horizontal Wind Model | Horizontal Wind Model 14

External Websites

NOAA National Geophysical Data Center

Horizontal Wind Model 14

Implement Horizontal Wind Model 14

Libraries:
Aerospace Blockset / Environment / Wind

Description

The Horizontal Wind Model 14 block implements the U.S. Naval Research Laboratory (HWM) routine to calculate the meridional and zonal components of the wind for a set of geographic coordinates: latitude, longitude, and altitude.

Limitation

For code generation, use this block only for targets whose type is int 32 or higher.

Ports

Input

First — geodetic latitude (μ), longitude (l), and geopotential altitude (h)
three-element vector of doubles

The input specifies the geodetic latitude (μ), longitude (l), and geopotential altitude (h) where the block implements the model.

Latitude and longitude values are in degrees.

The altitude value is in the units you selected in the **Units** parameter. Specify the altitude element as a value between 0 and 500 km. Values are held outside the range 0 to 500 km.

Second — day of year
scalar double

The input specifies the day of year in Coordinated Universal Time (UTC). The input specifies the day as a value between 1 and 366 (for a leap year). Values are wrapped within 1 to 366 days.

Third — elapsed seconds
scalar double

Contains elapsed seconds since midnight for the selected day, in UTC.

Fourth (Optional) — Ap index
scalar double

Contains the Ap index for the Universal Time (UT) when the block evaluates the model. Select the index from the NOAA National Geophysical Data Center, which contains 3 hour interval geomagnetic disturbance index values. If the Ap index value is greater than zero, the software takes into account magnetic effects during model evaluation.

Output

First — wind velocity vector
1-by-2 vector of doubles

The wind velocity vector contains the meridional and zonal wind components in that order.

Parameters

Units — input and output units
`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units for wind speed and velocity, specified as:

Units	Wind Speed	Wind Velocity
`Metric (MKS)`	Meters per second	Meters per second
`English (Velocity in ft/s)`	Feet per second	Feet per second
`English (Velocity in kts)`	Knots	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Model — horizontal wind model
`Quiet` (default) | `Total` | `Disturbance`

Select the horizontal wind model type for which to calculate the wind components.

- `Quiet`

  Calculate the horizontal wind model without the magnetic disturbances. For this model type, do not input an Ap index value.
- `Total`

  Calculate the combined effect of the quiet and magnetic disturbances. For this model type, input Ap index values greater than or equal to zero.
- `Disturbance`

  Calculate the effect of magnetic disturbances in the wind. For this model type, input Ap index values greater than or equal to zero.

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'Quiet'` | `'Total'` | `'Disturbance'`
Default: `'Quiet'`

Action for out-of-range input — block behavior
`Error` (default) | `Warning` | `None`

Specify the block behavior when the block inputs are out of range.

Value	Description
`Error` (default)	MATLAB returns an exception, and model simulation stops. For Accelerator and Rapid Accelerator modes, setting the action to `Error` has no effect and the model behaves as though the action is set to `None`.
`Warning`	Warning in the Diagnostic Viewer, and model simulation continues. For Accelerator and Rapid Accelerator modes, setting the action to `Warning` has no effect and the model behaves as though the action is set to `None`.
`None`	No action. The block imposes upper and lower limits on an input signal.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`

Version History

Introduced in R2016b

R2021b: Horizontal Wind Model 14 Block Possible Changed Returned Values

Behavior changed in R2021b

The Horizontal Wind Model 14 block now accepts:

- **day** port values that are decimal, negative, `0`, or greater than `366`.
- **sec** port values that are `0` or greater than `86400`.

As a result, the output values from this block might change from previous releases.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

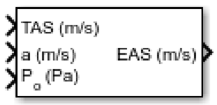
Horizontal Wind Model | Horizontal Wind Model 07

External Websites

NOAA National Geophysical Data Center

Ideal Airspeed Correction

Calculate equivalent airspeed (EAS), calibrated airspeed (CAS), or true airspeed (TAS) from each other



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Ideal Airspeed Correction block calculates one of these airspeeds from one of the other two airspeeds:

- Equivalent airspeed (EAS)
- Calibrated airspeed (CAS)
- True airspeed (TAS)

Limitations

This block assumes that the air flow is compressible dry air with constant specific heat ratio, γ

Ports

Input

TAS — True input airspeed
scalar

True input airspeed, specified as a scalar, in the units specified by the **Units** parameter.

Dependencies

To enable this port, set **Airspeed input** to TAS.

Data Types: `double`

EAS — Equivalent input airspeed
scalar

Equivalent input airspeed, specified as a scalar, in the units specified by the **Units** parameter.

Dependencies

To enable this port, set **Airspeed input** to EAS.

Data Types: `double`

CAS — Calibrated input airspeed
scalar

Calibrated input airspeed, specified as a scalar, in the units specified by the **Units** parameter.

Dependencies

To enable this port, set **Airspeed input** to EAS.

Data Types: `double`

a — Speed of sound
scalar

Speed of sound, specified as a scalar, in the units specified by the **Units** parameter.

Data Types: `double`

P_0 — Static pressure
scalar

Static pressure, specified as a scalar, in the units specified by the **Units** parameter.

Data Types: `double`

Output

EAS — Equivalent output airspeed
scalar

Equivalent output airspeed, returned as a scalar, in the units specified by the **Units** parameter.

Dependencies

To enable this port, set **Airspeed input** to TAS or CAS and **Airspeed output** to EAS.

Data Types: `double`

CAS — Calibrated output airspeed
scalar

Calibrated output airspeed, returned as a scalar, in the units specified by the **Units** parameter.

Dependencies

To enable this port, set **Airspeed input** to TAS or EAS and **Airspeed output** to CAS.

Data Types: `double`

TAS — True output airspeed
scalar

True output airspeed, returned as a scalar, in the units specified by the **Units** parameter.

Dependencies

To enable this port, set **Airspeed input** to CAS or EAS and **Airspeed output** to TAS.

Data Types: `double`

Parameters

Units — Units

Metric (MKS) (default) | English (Velocity in ft/s) | English (Velocity in kts)

Input and output units, specified as:

Units	Airspeed Input	Speed of Sound	Air Pressure	Airspeed Output
Metric (MKS)	Meters per second	Meters per second	Pascal	Meters per second
English (Velocity in ft/s)	Feet per second	Feet per second	Pound force per square inch	Feet per second
English (Velocity in kts)	Knots	Knots	Pound force per square inch	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Airspeed input — Airspeed input type

TAS (default) | EAS | CAS

Airspeed input type, specified as:

TAS	True airspeed
EAS	Equivalent airspeed
CAS	Calibrated airspeed

Programmatic Use
Block Parameter: `vel_in`
Type: character vector
Values: `'TAS'` | `'EAS'` | `'CAS'`
Default: `'TAS'`

Airspeed output — Airspeed output type

EAS (default) | CAS | TAS

Airspeed output type, specified as:

Airspeed Input	Airspeed Output
TAS	EAS (equivalent airspeed)
	CAS (calibrated airspeed)

Airspeed Input	Airspeed Output
EAS	TAS (true airspeed)
	CAS (calibrated airspeed)
CAS	TAS (true airspeed)
	EAS (equivalent airspeed)

Programmatic Use
Block Parameter: vel_out_tas, vel_out_cas, vel_out_eas, depending on the input velocity type, vel_in. For more information, see the airspeed output type table.
Type: character vector
Values: 'EAS' | 'CAS''TAS'
Default: 'EAS'

Method — Method for computing conversion factor

Table Lookup (default) | Equation

Method for computing the conversion factor, specified as:

Table Lookup	(Default) Generate output airspeed by looking up or estimating table values based on block inputs. If the **Subsonic airspeeds only** check box is selected, the Ideal Airspeed Correction block generates code that includes subsonic (Mach < 1) lookup table data. If the **Subsonic airspeeds only** check box is cleared, the Ideal Airspeed Correction block generates code that includes all (Mach < 5) lookup table data. Beyond Mach 5, the block uses the equation method. The Table Lookup method is not recommended for either of these instances: • Speed of sound less than 200 m/s or greater than 350 m/s. • Static pressure less than 1000 Pa or greater than 106,500 Pa. Using the Table Lookup method in these instances causes inaccuracies.
Equation	Compute output airspeed directly using block input values. Calculations involving supersonic airspeeds (greater than Mach 1) require an iterative computation. If the function does not find a solution within 30 iterations, it displays an error message. The block does not include lookup table data in generated code.

The Ideal Airspeed Correction block automatically uses the Equation method for any of these instances:

- Conversion with **Airspeed input** set to TAS and **Airspeed output** set to EAS.

- Conversion with **Airspeed input** set to EAS and **Airspeed output** set to TAS.
- Conversion when block input airspeed is greater than five times the speed of sound at sea level (approximately 1700 m/s).

Programmatic Use
Block Parameter: `method`
Type: character vector
Values: `'Table Lookup'` | `'Equation'`
Default: `'Table Lookup'`

Subsonic airspeeds only — Use with subsonic airspeed

`off` (default) | `on`

Select this check box to use this block only with subsonic airspeed (airspeeds less than Mach 1) applications. Selecting this check box may improve performance.

The block generates code as follows:

- If this check box is selected, the Ideal Airspeed Correction block generates code that includes subsonic (Mach < 1) lookup table data if **Method** is set to `Table Lookup`.

  Selecting this check box displays the **Action for out-of-range input** parameter.
- If this check box is cleared, the Ideal Airspeed Correction block generates code that includes all (Mach < 5) lookup table data if **Method** is set to `Table Lookup`.

Programmatic Use
Block Parameter: `SubOnly`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Action for out-of-range input — Out-of-range block behavior

None (default) | `Warning` | `Error`

Out-of-range block behavior, where airspeed is greater than Mach 1, specified as follows.

Value	Description
`None`	Does not display warning or error.
`Warning`	Displays warning and indicates that the airspeed is greater than Mach 1.
`Error`	Displays error and indicates that the airspeed is greater than Mach 1.

Dependencies

This parameter is enabled only if the **Subsonic airspeeds only** check box is selected.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`

Default: `'None'`

Version History

Introduced before R2006a

References

[1] Lowry, J. T., *Performance of Light Aircraft*, AIAA Education Series, Washington, DC, 1999.

[2] *Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986.

[3] Gracey, William, *Measurement of Aircraft Speed and Altitude*, NASA Reference Publication 1046, 1980.

Extended Capabilities

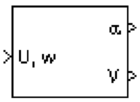
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

COESA Atmosphere Model | ISA Atmosphere Model | Lapse Rate Model | Non-Standard Day 210C | Non-Standard Day 310

Incidence & Airspeed

Calculate incidence and airspeed



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Incidence & Airspeed block supports the 3DoF equations of motion model by calculating the angle between the velocity vector and the body, and also the total airspeed from the velocity components in the body-fixed coordinate frame.

$$\alpha = \operatorname{atan}\left(\frac{w}{u}\right)$$

$$V = \sqrt{u^2 + w^2}$$

Ports

Input

U, w — Velocity
two-element vector

Velocity of the body, specified as a two-element vector, resolved into the body-fixed coordinate frame.

Data Types: `double`

Output

α — Incidence angle
scalar

Incidence angle, returned as a scalar, in radians.

Data Types: `double`

V — Airspeed
scalar

Airspeed of the body, returned as a scalar.

Data Types: `double`

Version History

Introduced before R2006a

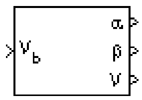
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Incidence, Sideslip, & Airspeed

Incidence, Sideslip, & Airspeed

Calculate incidence, sideslip, and airspeed



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Incidence, Sideslip, & Airspeed block calculates the angles between the velocity vector and the body, and also the total airspeed from the velocity components in the body-fixed coordinate frame. For the equations used in the calculation, see “Algorithms” on page 5-501.

Ports

Input

V_b — Velocity of body
three-element vector

Velocity of the body, specified as a three-element vector, resolved into the body-fixed coordinate frame.

Data Types: `double`

Output

α — Incidence angle
scalar

Incidence angle, returned as a scalar, in radians.

Data Types: `double`

β — Sideslip angle
scalar

Sideslip angle, returned as a scalar, in radians.

Data Types: `double`

V — Airspeed
scalar

Airspeed of the body, returned as a scalar.

Data Types: `double`

Algorithms

To calculate the angles between the velocity vector and the body, and the total airspeed, the block uses these equations:

$$\alpha = \tan^{-1}\left(\frac{w}{u}\right)$$

$$\beta = \sin^{-1}\left(\frac{v}{V}\right) = \tan^{-1}\left(\frac{v}{\sqrt{u^2 + w^2}}\right)$$

The block uses the `tan`$^{-1}$ formulation for β to prevent division by `0`.

$$V = \sqrt{u^2 + v^2 + w^2}$$

Version History

Introduced before R2006a

R2022b: Updated to Reduce Divide by Zero Issues

Behavior changed in R2022b

The Incidence, Sideslip, & Airspeed block now reduces the potential for divide by zero issues caused by aircraft hovering without forward movement. In previous releases, hovering without forward movement might cause divide by zero issues.

Extended Capabilities

C/C++ Code Generation

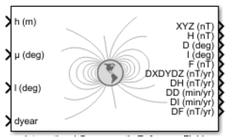
Generate C and C++ code using Simulink® Coder™.

See Also

Incidence & Airspeed

International Geomagnetic Reference Field

Calculate Earth magnetic field and secular variation using International Geomagnetic Reference Field



Libraries:
Aerospace Blockset / Environment / Gravity

Description

The International Geomagnetic Reference Field block calculates the Earth magnetic field and secular variation using the selected International Geomagnetic Reference Field generation. It calculates the Earth magnetic field and secular variation at a position and time using the selected International Geomagnetic Reference Field generation.

Limitations

- This block is valid between the heights of -1000 m and 5.6 Earth radii (35,717,567.2 m).
- This block is valid for these year ranges:
  - IGRF-13 model — 1900 and 2025
  - IGRF-12 model — 1900 and 2020
  - IGRF-11 model — 1900 and 2015
- If the decimal year is outside the valid range for a generation, the International Geomagnetic Reference Field block linearly extrapolates the magnetic field to the out-of-range decimal year.
- For additional limitations, see :

  `IGRF Health Warning, Errors, and Limitations`

Ports

Input

h — Height
scalar

Height, specified as a scalar, in selected units.

Data Types: `double`

μ (deg) — Latitude
scalar

Latitude, specified as a scalar in degrees. This block accepts latitude values greater than 90 and less than -90.

Data Types: `double`

l (deg) — Longitude
scalar

Longitude, specified as a scalar, in degrees. This block accepts ranges greater than 180 and less than -180.

Data Types: `double`

dyear — Desired year
scalar

Desired year in a decimal format to include any fraction of the year that has already passed. The value is the current year plus the number of days that have passed in this year divided by 365. To calculate the decimal year, `dyear`, for March 21, 2015:

```
dyear=decyear('21-March-2015','dd-mmm-yyyy')
```

Dependencies

To enable this port, select **Input decimal year**.

Data Types: `double`

Output

XYZ — Magnetic field
vector

Magnetic field, returned as a vector, in selected units. The components of this vector are in the north-east-down (NED) reference frame.

Data Types: `double`

H — Horizontal intensity
scalar

Horizontal intensity, returned as a scalar, in selected units.

Data Types: `double`

D — Declination
scalar

Declination, returned as a scalar, in degrees.

Data Types: `double`

I — Inclination
scalar

Inclination, returned as a scalar, in degrees.

Data Types: `double`

F — Total intensity
scalar

Total intensity, returned as a scalar, in selected units.

Data Types: `double`

DXDYDZ — Secular variation of magnetic field
vector

Secular variation of magnetic field, returned as a vector in selected units per year.

Dependencies

To enable this port, select **Output secular variation**.

Data Types: `double`

DH — Secular variation of horizontal intensity
scalar

Secular variation of horizontal intensity, returned as a scalar, in selected units per year.

Dependencies

To enable this port, select **Output secular variation**.

Data Types: `double`

DD — Secular variation of declination
scalar

Secular variation of declination, returned as a scalar, in minutes per year.

Dependencies

To enable this port, select **Output secular variation**.

Data Types: `double`

DI — Secular variation of inclination
scalar

Secular variation of inclination, returned as a scalar, in minutes per year.

Dependencies

To enable this port, select **Output secular variation**.

Data Types: `double`

DF — Secular variation of total intensity
scalar

Secular variation of total intensity, returned as a scalar, in selected units per year.

Dependencies

To enable this port, select **Output secular variation**.

Data Types: `double`

Parameters

Generation — International Geomagnetic Reference Field generation

IGRF-13 (default) | IGRF-11 | IGRF-12

International Geomagnetic Reference Field generation, selected from IGRF-13, IGRF-12, or IGRF-11.

Programmatic Use
Block Parameter: generation
Type: character vector
Values: 'IGRF-13' | 'IGRF-11' | 'IGRF-12'
Default: 'IGRF-13'

Data Types: char | string

Units — Units

Metric (MKS) (default) | English

Parameter and output units, specified as:

Units	Height
Metric (MKS)	Meters
English	Feet

Programmatic Use
Block Parameter: units
Type: character vector
Values: 'Metric (MKS)' | 'English'
Default: 'Metric (MKS)'

Input decimal year — Desired year

on (default) | off

- To specify the decimal year with an input port, select this check box.
- To specify the decimal year using the values of **Month**, **Day**, and **Year**, clear this check box.

Dependencies

To enable **Month**, **Day**, and **Year**, clear this parameter.

Programmatic Use
Block Parameter: time_in
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Month — Input month

January (default) | February | March | April | May | June | July | August | September | October | November | December

Month to calculate decimal year.

Dependencies

To enable this parameter, clear **Input decimal year**.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Input day

1 (default) | 1 to 31

Day to calculate decimal year.

Dependencies

To enable this parameter, clear **Input decimal year**.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` to `'31'`
Default: `'1'`

Year — Input year

`2020` (default) | `1900` to `2020`

Year to calculate decimal year, specified as `1900` to `2020`.

Dependencies

To enable this parameter, clear **Input decimal year**.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: any year
Default: `'2020'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`

Default: `'Warning'`

Output secular variance — Secular variances

on (default) | `off`

Select this check box to enable the output of secular variances (annual rate of change) with nonsecular variances. Otherwise, clear this check box.

Secular Variance	Description
Magnetic Field	Magnetic field vector, in nanotesla (nT). *Z* is the vertical component (+ve down)
Horizontal Intensity	Horizontal intensity, in nanotesla (nT)
Declination	Declination, in degrees (+ve east)
Inclination	Inclination, in degrees (+ve down)
Total Intensity	Total intensity, in nanotesla (nT)
SV Magnetic Field	Secular variation of magnetic field
SV Horizontal Intensity	Secular variation of horizontal intensity
SV Declination	Secular variation of declination, the angle between true north and the magnetic field vector (positive eastward)
SV Inclination	Secular variation of inclination, the angle between the horizontal plane and the magnetic field vector (positive downward)
SV Total Intensity	Secular variation of total intensity

Clear this check box to enable just the nonsecular variances:

- Magnetic Field
- Horizontal Intensity
- Declination
- Inclination
- Total Intensity

Programmatic Use
Block Parameter: `sv_out`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

World Magnetic Model

External Websites

IGRF Health Warning, Errors, and Limitations

Interpolate Matrix(x)

Return interpolated matrix for given input

Libraries:
Aerospace Blockset / GNC / Control

Description

The Interpolate Matrix(x) block interpolates a one-dimensional array of matrices. The block assumes a one-dimensional array as defined in “Algorithms” on page 5-510.

The matrix to be interpolated must be three dimensional, the first two dimensions corresponding to the matrix at each value of *x*. For example, if you have three matrices *A*, *B*, and *C* defined at `x = 0`, `x = 0.5`, and `x = 1.0`, then the input matrix is given by

```
matrix(:,:,1) = A;

matrix(:,:,2) = B;

matrix(:,:,3) = C;
```

Limitations

This block must be driven from the Prelookup block.

Ports

Input

x_k — Interpolation index *i*
scalar

Interpolation index *i*, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

x_f — Interpolation fraction
scalar

Interpolation fraction λ, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

Matrix(x) — Interpolated matrix
matrix

Interpolated matrix, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Matrix to interpolate — Matrix

`matrix` (default)

Matrix to be interpolated, with three indices and the third index labeling the interpolating values of *x*.

Programmatic Use
Block Parameter: `matrix`
Type: character vector
Values: matrix
Default: `'matrix'`

Algorithms

This one-dimensional case assumes a matrix *M* is defined at a discrete number of values of an independent variable

$x = [\ x_1 x_2 x_3 \ldots x_i x_{i+1} \ldots x_n\ ]$.

Then for $x_i < x < x_{i+1}$, the block output is given by

$$(1 - \lambda)M(x_i) + \lambda M(x_{i+1})$$

where the interpolation fraction is defined as

$$\lambda = (x - x_i)/(x_{i+1} - x_i)$$

Version History

Introduced before R2006a

Extended Capabilities

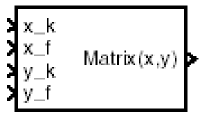
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

on page 5-2 | 1D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 1D Self-Conditioned [A(v),B(v),C(v),D(v)] | Interpolate Matrix(x,y) | Interpolate Matrix(x,y,z)

Interpolate Matrix(x,y)

Return interpolated matrix for given inputs



Libraries:
Aerospace Blockset / GNC / Control

Description

The Interpolate Matrix(x,y) block interpolates a two-dimensional array of matrices. In two-dimensional cases, the interpolation is carried out first on *x* and then *y*. For more information, see "Algorithms" on page 5-512.

The matrix to be interpolated must be four-dimensional, the first two dimensions corresponding to the matrix at each value of *x* and *y*. For example, if you have four matrices *A*, *B*, *C*, and *D* defined at `(x = 0.0,y = 1.0)`, `(x = 0.0,y = 3.0)`, `(x = 1.0,y = 1.0)` and `(x = 1.0,y = 3.0)`, then the input matrix is given by

```
matrix(:,:,1,1) = A;

matrix(:,:,1,2) = B;

matrix(:,:,2,1) = C;

matrix(:,:,2,2) = D;
```

Limitations

This block must be driven from the Prelookup block.

Ports

Input

x_k — First interpolation index
scalar

First interpolation index *i*, specified as a scalar and vector.

Data Types: `double`

x_f — First interpolation fraction
scalar

First interpolation fraction λ_x, specified as a scalar

Data Types: `double`

y_k — Second interpolation index
scalar

Second interpolation index j, specified as a scalar.

Data Types: `double`

y_f — Second interpolation fraction
scalar

Second interpolation fraction λ_y, specified as a scalar.

Data Types: `double`

Output

Matrix(x,y) — Interpolated matrix
matrix

Interpolated matrix, specified as a matrix.

Data Types: `double`

Parameters

Matrix to interpolate — Matrix

`matrix` (default)

Matrix to be interpolated, with four indices and the third and fourth indices labeling the interpolating values of x and y.

Programmatic Use
Block Parameter: `matrix`
Type: character vector
Values: matrix
Default: `'matrix'`

Algorithms

This two-dimensional case assumes the matrix is defined as a function of two independent variables, $\boldsymbol{x} = [\ x_1 x_2 x_3 \ldots\ x_i x_{i+1}\ \ldots\ x_n]$ and $\boldsymbol{y} = [\ y_1 y_2 y_3\ \ldots\ y_j y_{j+1}\ \ldots\ y_m]$. For given values of x and y, four matrices are interpolated. Then for $x_i < x < x_{i+1}$ and $y_j < y < y_{j+1}$, the output matrix is given by

$$(1-\lambda_y)[(1-\lambda_x)M(x_i, y_j) + \lambda_x M(x_{i+1}, y_j)] +$$
$$\lambda_y[(1-\lambda_x)M(x_i, y_{j+1}) + \lambda_x M(x_{i+1}, y_{j+1})]$$

where the two interpolation fractions are denoted by

$$\lambda_x = (x - x_i)/(x_{i+1} - x_i)$$

and

$$\lambda_y = (y - y_j)/(y_{j+1} - y_j)$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

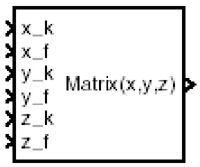
Generate C and C++ code using Simulink® Coder™.

See Also

2D Controller [A(v),B(v),C(v),D(v)] | 2D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | Interpolate Matrix(x) | Interpolate Matrix(x,y,z) | Prelookup

Interpolate Matrix(x,y,z)

Return interpolated matrix for given inputs



Libraries:
Aerospace Blockset / GNC / Control

Description

The Interpolate Matrix(x,y,z) block interpolates a three-dimensional array of matrices.

This three-dimensional case assumes the matrix is defined as a function of three independent variables:

$x = [\ x_1\ x_2\ x_3 \dots x_i\ x_{i+1} \dots x_n\ ]$

$y = [\ y_1\ y_2\ y_3 \dots y_j\ y_{j+1} \dots y_m\ ]$

$z = [\ z_1\ z_2\ z_3 \dots z_k\ z_{k+1} \dots z_p\ ]$

For given values of x, y, and z, eight matrices are interpolated. Then for

$x_i < x < x_{i+1}$

$y_j < y < y_{j+1}$

$z_k < z < z_{k+1}$

the output matrix is given by

$$\begin{aligned}(1-\lambda_z)\Big\{(1-\lambda_y)\big[(1-\lambda_x)M(x_i, y_j, z_k) + \lambda_x M(x_{i+1}, y_j, z_k)\big] \\ +\lambda_y\big[(1-\lambda_x)M(x_i, y_{j+1}, z_k) + \lambda_x M(x_{i+1}, y_{j+1}, z_k)\big]\Big\} \\ +\lambda_z\Big\{(1-\lambda_y)\big[(1-\lambda_x)M(x_i, y_j, z_{k+1}) + \lambda_x M(x_{i+1}, y_j, z_{k+1})\big] \\ +\lambda_y\big[(1-\lambda_x)M(x_i, y_{j+1}, z_{k+1}) + \lambda_x M(x_{i+1}, y_{j+1}, z_{k+1})\big]\Big\}\end{aligned}$$

where the three interpolation fractions are denoted by

$$\begin{aligned}\lambda_x &= (x - x_i)/(x_{i+1} - x_i) \\ i_y &= (y - y_j)/(y_{j+1} - y_j) \\ \lambda_z &= (z - z_k)/(z_{k+1} - z_k)\end{aligned}$$

In the three-dimensional case, the interpolation is carried out first on x, then y, and finally z.

The matrix to be interpolated should be five-dimensional, the first two dimensions corresponding to the matrix at each value of x, y, and z. For example, if you have eight matrices A, B, C, D, E, F, G, and H defined at the following values of x, y, and z, then the corresponding input matrix is given by

`(x = 0.0,y = 1.0,z = 0.1)`	`matrix(:,:,1,1,1) = A;`
`(x = 0.0,y = 1.0,z = 0.5)`	`matrix(:,:,1,1,2) = B;`
`(x = 0.0,y = 3.0,z = 0.1)`	`matrix(:,:,1,2,1) = C;`
`(x = 0.0,y = 3.0,z = 0.5)`	`matrix(:,:,1,2,2) = D;`
`(x = 1.0,y = 1.0,z = 0.1)`	`matrix(:,:,2,1,1) = E;`
`(x = 1.0,y = 1.0,z = 0.5)`	`matrix(:,:,2,1,2) = F;`
`(x = 1.0,y = 3.0,z = 0.1)`	`matrix(:,:,2,2,1) = G;`
`(x = 1.0,y = 3.0,z = 0.5)`	`matrix(:,:,2,2,2) = H;`

Limitations

This block must be driven from the Prelookup block.

Ports

Input

x_k — First interpolation index
scalar

First interpolation index i, specified as a scalar.

Data Types: `double`

x_f — First interpolation fraction
scalar

First interpolation fraction λ_x, specified as a scalar .

Data Types: `double`

y_k — Second interpolation index
scalar

Second interpolation index j, specified as a scalar.

Data Types: `double`

y_f — Second interpolation fraction
scalar

Second interpolation fraction λ_y, specified as a scalar.

Data Types: `double`

z_k — Third interpolation index
scalar

Third interpolation index k, specified as a scalar.

Data Types: `double`

z_f — Third interpolation fraction
scalar

Third interpolation fraction λ_z, specified as a scalar.

Data Types: `double`

Output

Matrix(x,y,z) — Interpolated matrix
matrix

Interpolated matrix, specified as a matrix.

Data Types: `double`

Parameters

Matrix to interpolate — Matrix to interpolate

`matrix` (default)

Matrix to be interpolated, with five indices and the third, fourth, and fifth indices labeling the interpolating values of x, y, and z.

Programmatic Use
Block Parameter: `matrix`
Type: character vector
Values: matrix
Default: `'matrix'`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

3D Controller [A(v),B(v),C(v),D(v)] | 3D Observer Form [A(v),B(v),C(v),F(v),H(v)] | 3D Self-Conditioned [A(v),B(v),C(v),D(v)] | Interpolate Matrix(x) | Interpolate Matrix(x,y) | Prelookup

Invert 3x3 Matrix

Compute inverse of 3-by-3 matrix

Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Invert 3x3 Matrix block computes the inverse of 3-by-3 matrix.

If $det(A) = 0$, an error occurs and the simulation stops.

Ports

Input

Port_1 — Input matrix
3-by-3 matrix

Input matrix to be inverted, specified as a 3-by-3 matrix.

Data Types: `double`

Output

Port_1 — Matrix inverse
3-by-3 matrix

Matrix inverse of input matrix, returned as a 3-by-3 matrix.

Data Types: `double`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

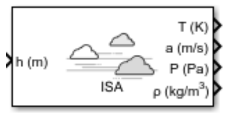
Generate C and C++ code using Simulink® Coder™.

See Also

Adjoint of 3x3 Matrix | Create 3x3 Matrix | Determinant of 3x3 Matrix

ISA Atmosphere Model

Implement International Standard Atmosphere (ISA)



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The ISA Atmosphere Model block implements the mathematical representation of the International Standard Atmosphere (ISA) values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude.

The ISA Atmosphere Model, Extended ISA model, and Lapse Rate Model blocks are identical blocks. The ISA Atmosphere Model block has three implementations:

- The default ISA model implements the mathematical representation of the ISA values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude between sea level (0 m) and tropopause (20000 m).
- The extended ISA model implements the mathematical representation of the ISA values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude between -5000 m and mesopause (84852 m).
- The Lapse rate model implements custom lapse rate atmosphere model using the input parameters.

The ISA Atmosphere Model block icon displays the input and output port labels in metric units.

Limitations

- In the default model, the block limits the implementation to the geopotential altitude of 0 m (minimum) and 20000 m (maximum). In the extended model, the limits are -5000 meters (minimum) and 84852 meters (maximum). The limits are inclusive.
- Density and speed of sound are calculated using a perfect gas relationship.

Ports

Input

h (m) — Geopotential height
scalar | *m*-by-*n* array

Geopotential height, specified as a scalar or *m*-by-*n* array.

Data Types: `double`

Output

T (K) — Temperature
scalar | *m*-by-*n* array

Temperature, returned as a scalar or *m*-by-*n* array, in K.

Data Types: `double`

a (m/s) — Speed of sound
scalar | *m*-by-*n* array

Speed of sound, returned as a scalar or *m*-by-*n* array, in m/s.

Data Types: `double`

P (Pa) — Air pressure
scalar | *m*-by-*n* array

Air pressure, returned as a scalar or *m*-by-*n* array, in Pa.

Data Types: `double`

ρ (kg/m^3) — Densities
scalar | *m*-by-*n* array

Densities, returned as a scalar or an *m*-by-*n* array, in kg/m^3.

Data Types: `double`

ν (m^2/s) — Kinematic viscosity
scalar | *m*-by-*n* array

Kinematic viscosity, returned as a scalar or *m*-by-*n* array, in m^2/s.

Dependencies

Selecting the **Output kinematic and dynamic viscosity** check box adds the **ν (m^2/s)** output port.

Data Types: `double`

μ (N s/m^2) — Dynamic viscosity
scalar | *m*-by-*n* array

Dynamic viscosity, returned as a scalar or *m*-by-*n* array, in N s/m^2.

Dependencies

Selecting the **Output kinematic and dynamic viscosity** check box adds the **μ (N s/m^2)** output port.

Data Types: `double`

Parameters

Use extended ISA model — Range of implementation

`off` (default) | `on`

Mathematical representation of the International Standard Atmosphere model.

- To implement the mathematical representation of the ISA values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude between the sea level and the tropopause, clear this checkbox. For input heights outside the limits, the output values are held constant.
- To implement the mathematical representation of the ISA values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude between -5000 meters and the mesopause, select this check box. For input heights outside the limits, the output values are held constant if the out-of-range action input is set as `None` or `Warning`.

Dependencies

Selecting this check box removes the **Change atmospheric parameters** check box.

Programmatic Use
Block Parameter: `useExtended`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

S — Sutherland temperature
110.4 K (default) | scalar

Sutherland temperature in Kelvin, specified as a scalar.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** and **Change atmospheric parameters** are e.

T_{ref} — Reference temperature
273.15 K (default) | scalar

Reference temperature in Kelvin, specified as a scalar.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** is selected.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** and **Change atmospheric parameters** are e.

μ_{ref} — Reference dynamic viscosity
1.716e-5 kg/ms (default) | scalar

Reference dynamic viscosity in kilograms per meter second, specified as a scalar.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** and **Change atmospheric parameters** are e.

Output kinematic and dynamic viscosity — Option to output kinematic and dynamic viscosity

`off` (default) | `on`

Option to output kinematic and dynamic viscosity.

Dependencies

Selecting this check box:

- Adds the **ν (m²/s)** output port.
- Adds the **μ (N s/m²)** output port.

Dependencies

If selected simultaneously with the **Change atmospheric parameters** checkbox, adds the **Viscosity parameters** tab.

Programmatic Use
Block Parameter: `viscosityOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Action for out-of-range input — Out-of-range block behavior

`Warning` (default) | `Error` | `None`

Out-of-range block behavior, specified as follows:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error`	Error in the Diagnostic Viewer, model simulation stops.

Dependencies

To enable this parameter, select **Use extended ISA model**.

If this parameter is not visible, the default action is `None`.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Change atmospheric parameters — Customize parameters

`off` (default) | `on`

Customize various atmospheric parameters to be different from the ISA values. Selecting this check box converts the block from ISA Atmosphere Model to Lapse Rate Model.

Dependencies

Selecting this check box enables the **Atmospheric parameters** tab with these parameters:

- **Acceleration due to gravity (m/s^2)**
- **Ratio of specific heats**
- **Characteristic gas constant (J/Kg/K)**
- **Lapse rate (K/m)**
- **Height of troposphere (m)**
- **Height of tropopause (m)**
- **Air density at mean sea level (Kg/m^3)**
- **Ambient pressure at mean sea level (N/m^2)**
- **Lowest altitude (m)**

Selecting in conjunction with **Output kinematic and dynamic viscosity** enables the **Viscosity parameters** tab with these parameters:

- **S**
- $\mathbf{T_{ref}}$
- $\boldsymbol{\mu_{ref}}$

Selecting this check box removes the **Use extended ISA model** check box.

Programmatic Use
Block Parameter: `custom`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Acceleration due to gravity (m/s^2) — Acceleration

`9.80665` (default) | scalar

Acceleration from gravity (g). in m/s^2, specified as double scalar.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `g`
Type: character vector
Values: double scalar
Default: `9.80665`

Ratio of specific heats — Ratio of heats

`1.4` (default) | scalar

Ratio of specific heats γ, specified as a double value.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `gamma`
Type: character vector
Values: double scalar
Default: `1.4`

Characteristic gas constant (J/Kg/K) — Gas constant

`287.0531` (default) | scalar

Characteristic gas constant (R), specified as double scalar, in J/Kg/K.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `287.0531`

Lapse rate (K/m) — Lapse rate

`0.0065` (default) | scalar

Lapse rate of the troposphere, specified as double scalar, in K/m.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `L`
Type: character vector
Values: double scalar
Default: `0.0065`

Height of troposphere (m) — Troposphere height

`11000` (default) | scalar

Height of the troposphere (range of decreasing temperatures), specified as double scalar, in m.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `h_trop`
Type: character vector
Values: double scalar
Default: `11000`

Height of tropopause (m) — Tropopause height

`20000` (default) | scalar

Height of the tropopause (range of constant temperature), specified as double scalar, in m.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use

Block Parameter: `h_strat`
Type: character vector
Values: double scalar
Default: `20000`

Air density at mean sea level (Kg/m^3) — Air density

`1.225` (default) | scalar

Air density at mean sea level, specified as double scalar, in Kg/m^3.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use

Block Parameter: `rho0`
Type: character vector
Values: double scalar
Default: `1.225`

Ambient pressure at mean sea level (N/m^2) — Ambient pressure

`101325` (default) | scalar

Ambient pressure at mean sea level, specified as double scalar, in N/m^2.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use

Block Parameter: `P0`
Type: character vector
Values: double scalar
Default: `101325`

Ambient temperature at mean sea level (K) — Ambient temperature

`288.15` (default) | scalar

Ambient temperature at mean sea level (T_0), specified as double scalar, in K.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use

Block Parameter: `K`
Type: character vector

Values: double scalar
Default: `101325`

Lowest altitude (m) — Lowest altitude

`0` (default) | scalar

Lowest altitude above which temperature and pressure lapse, specified as double scalar, in m. **Lowest altitude (m)** must be below **Height of tropopause**.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use

Block Parameter: `h0`
Type: character vector
Values: double scalar
Default: `0`

Algorithms

The dynamic viscosity is computed using [1]:

$$\mu = \frac{\beta \bullet T^{3/2}}{T + S}$$

The kinematic viscosity is computed using [1]:

$$\eta = \frac{\mu}{\rho}$$

where:

- β — A constant dependent on the reference temperature and dynamic viscosity.
- T — Kelvin kinetic temperature.
- S — Sutherland constant.
- ρ — Mass density of air.

Version History

Introduced before R2006a

R2023b: ISA Atmosphere Model Block Changes

The ISA Atmosphere Model block now supports geopotential altitude inputs between and including -5000 meters and the mesopause, 84,852 meters. In previous releases, ISA Atmosphere Model supported geopotential altitudes between and including sea level and the tropopause, 20,000 meters.

To support the new capability, the block has these changes:

- **Use extended ISA model** parameter:

  - To implement the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude between the sea level and the tropopause, clear this check box.
  - To implement the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude between -5000 meters and the mesopause, select this check box.
- **Action for out-of-range input** parameter, which lets you specify the action for out-of-range input when the **Use extended ISA model** check box is selected.
- **Output kinematic and dynamic viscosity** parameter:
  - To return a numeric array of *m*-by-*n* values of kinematic viscosities, in m^2/s, in the **ν** output port.
  - To return a numeric array of *m*-by-*n* values of dynamic viscosities, in N s/m^2, in the **μ** output port.

R2024a: ISA Atmosphere Model Block Changes

Use the **Output kinematic and dynamic viscosity** checkbox and edit the **Viscosity parameters values** to compute the viscosity of any gas in the ISA Atmosphere Model block.

References

[1] *U.S. Standard Atmosphere.*, Washington, D.C.: U.S. Government Printing Office, 1976.

Extended Capabilities

C/C++ Code Generation

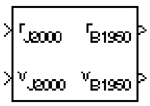
Generate C and C++ code using Simulink® Coder™.

See Also

COESA Atmosphere Model | CIRA-86 Atmosphere Model | Lapse Rate Model

Julian Epoch to Besselian Epoch

Transform position and velocity components from Standard Julian Epoch (J2000) to discontinued Standard Besselian Epoch (B1950)



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Julian Epoch to Besselian Epoch block transforms two 3-by-1 vectors of Julian Epoch position $(\bar{r}_{J2000})$, and Julian Epoch velocity $(\bar{v}_{J2000})$ into Besselian Epoch position $(\bar{r}_{B1950})$, and Besselian Epoch velocity $(\bar{v}_{B1950})$. For more information on the transformation, see "Algorithms" on page 5-528.

Ports

Input

r_{J2000} — Position
3-by-1 vector

Position in Standard Julian Epoch (J2000), specified as a 3-by-1 vector.

Data Types: `double`

v_{J2000} — Velocity
3-by-1 vector

Velocity in Standard Julian Epoch (J2000), specified as a 3-by-1 vector.

Data Types: `double`

Output

r_{B1950} — Position
3-by-1 vector

Position in Standard Besselian Epoch (B1950), returned as a 3-by-1 vector.

Data Types: `double`

v_{B1950} — Velocity
3-by-1 vector

Velocity in Standard Besselian Epoch (B1950), returned as a 3-by-1 vector.

Data Types: `double`

Algorithms

The transformation is calculated using:

$$\begin{bmatrix} \bar{r}_{B1950} \\ \bar{v}_{B1950} \end{bmatrix} = \begin{bmatrix} \overline{M}_{rr} & \overline{M}_{vr} \\ \overline{M}_{rv} & \overline{M}_{vv} \end{bmatrix}^T \begin{bmatrix} \bar{r}_{J2000} \\ \bar{v}_{J2000} \end{bmatrix},$$

where

$$\left(\overline{M}_{rr}, \overline{M}_{vr}, \overline{M}_{rv}, \overline{M}_{vv}\right)$$

are defined as:

$$\overline{M}_{rr} = \begin{bmatrix} 0.9999256782 & -0.0111820611 & -0.0048579477 \\ 0.0111820610 & 0.9999374784 & -0.0000271765 \\ 0.0048579479 & -0.0000271474 & 0.9999881997 \end{bmatrix}$$

$$\overline{M}_{vr} = \begin{bmatrix} 0.00000242395018 & -0.00000002710663 & -0.00000001177656 \\ 0.00000002710663 & 0.00000242397878 & -0.00000000006587 \\ 0.00000001177656 & -0.00000000006582 & 0.00000242410173 \end{bmatrix}$$

$$\overline{M}_{rv} = \begin{bmatrix} -0.000551 & -0.238565 & 0.435739 \\ 0.238514 & -0.002667 & -0.008541 \\ -0.435623 & 0.012254 & 0.002117 \end{bmatrix}$$

$$\overline{M}_{vv} = \begin{bmatrix} 0.99994704 & -0.01118251 & -0.00485767 \\ 0.01118251 & 0.99995883 & -0.00002718 \\ 0.00485767 & -0.00002714 & 1.00000956 \end{bmatrix}$$

Version History

Introduced before R2006a

References

[1] "Supplement to Department of Defense World Geodetic System 1984 Technical Report: Part I - Methods, Techniques and Data Used in WGS84 Development," DMA TR8350.2-A.

Extended Capabilities

C/C++ Code Generation

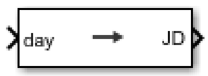
Generate C and C++ code using Simulink® Coder™.

See Also

Besselian Epoch to Julian Epoch

Julian Date Conversion

Calculate Julian date or modified Julian date



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Julian Date Conversion block converts the specified date to the Julian date or modified Julian date.

Limitations

- This block is valid for all common era (CE) dates in the Gregorian calendar.
- The calculation of Julian date does not take into account leap seconds.

Ports

Input

day — Clock source
scalar | array

Clock source for model simulation, specified as a scalar or array.

Dependencies

The presence and label of this port depends on the **Time increment** parameter.

Port	Time increment Setting
day	`Day`
hour	`Hour`
min	`Min`
sec	`Sec`
No inport port	`None`

Data Types: `double`

Output

JD — Julian date
scalar | array

Julian date, returned as a scalar or array.

Dependencies

Data Types: `double`

Parameters

Year — Year

`2013` (default) | double, whole number, greater than 1

Year, specified as a scalar, to calculate the Julian date.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: double, greater than 1
Default: `'2013'`

Month — Month

`January` (default) | `February` | `March` | `April` | `May` | `June` | `July` | `August` | `September` | `October` | `November` | `December`

Month to calculate the Julian date. From the list, select the month from `January` to `December`.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Day

`1` (default) | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10` | `11` | `12` | `13` | `14` | `15` | `16` | `17` | `18` | `19` | `20` | `21` | `22` | `23` | `24` | `25` | `26` | `27` | `28` | `29` | `30` | `31`

Day to calculate the Julian date. From the list, select the day from `1` to `31`.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'` | `'10'` | `'11'` | `'12'` | `'13'` | `'14'` | `'15'` | `'16'` | `'17'` | `'18'` | `'19'` | `'20'` | `'21'` | `'22'` | `'23'` | `'24'` | `'25'` | `'26'` | `'27'` | `'28'` | `'29'` | `'30'` | `'31'`
Default: `'1'`

Hour — Hour

`0` (default) | double, whole number, `0` to `24`

Hour used to calculate the Julian date. Enter a value from `0` to `24`.

Programmatic Use
Block Parameter: `hour`
Type: character vector
Values: double, whole number, `0` to `24`
Default: `'0'`

Minutes — Minutes

0 (default) | double, whole number, 0 to 60

Minutes to calculate the Julian date. Enter a number from 0 to 60.

Programmatic Use
Block Parameter: min
Type: character vector
Values: double, whole number, 0 to 60
Default: '0'

Seconds — Seconds

0 (default) | double, whole number, 0 to 60

Specify the seconds used to calculate the Julian date. Enter a number from 0 to 60.

Programmatic Use
Block Parameter: sec
Type: character vector
Values: double, whole number, 0 to 60
Default: '0'

Calculate modified Julian date — Modified Julian data

off (default) | on

Select this check box to calculate the modified Julian date (MJD) for corresponding elements of the year, month, day, hour, minute, and second.

Dependencies

Selecting this check box changes the output port label to MJD. Clearing this check box changes the output port label to JD.

Programmatic Use
Block Parameter: modflag
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Time increment — Time increment

Day (default) | Hour | Min | Sec | None

Time increment between the specified date and the desired model simulation time. The block adjusts the calculated Julian date to take into account the time increment from model simulation. For example, selecting Day and connecting a simulation timer to the port means that each time increment unit is one day and the block adjusts its calculation based on that simulation time.

If you select None, the calculated Julian date does not take into account the model simulation time. Selecting this option removes the first block input.

Dependencies

This parameter controls the presence and label of output port.

Time increment Setting	Port
Day	day
Hour	hour
Min	min
Sec	sec
None	No inport port

Programmatic Use
Block Parameter: deltaT
Type: character vector
Values: 'Day' | 'Hour' | 'Min' | 'Sec' | 'None'
Default: 'Day'

Action for out-of-range input — Out-of-range block behavior

None (default) | Warning | Error

Out-of-range block behavior, specified as follows.

Action	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: errorflag
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'Error'

Version History
Introduced in R2013b

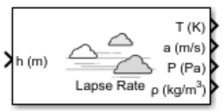
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
juliandate

Lapse Rate Model

Implement lapse rate model for atmosphere



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The Lapse Rate Model block implements the mathematical representation of the lapse rate atmospheric equations for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude. You can customize this atmospheric model by specifying atmospheric properties.

The ISA Atmosphere Model and Lapse Rate Model blocks are identical blocks. When configured for ISA Atmosphere Model, the block implements ISA values. When configured for Lapse Rate Model, the block implements the mathematical representation of lapse rate atmospheric equations.

The Lapse Rate Model block icon displays the input and output metric units.

Limitations

- Below the geopotential altitude of 0 km and above the geopotential altitude of the tropopause, temperature and pressure values are held.
- Density and speed of sound are calculated using a perfect gas relationship.

Ports

Input

h (m) — Geopotential height
scalar | array

Geopotential height, specified as a scalar or array.

Data Types: `double`

Output

T (K) — Temperature
scalar | array

Temperature, returned as a scalar or array, in K.

Data Types: `double`

a (m/s) — Speed of sound
scalar | array

Speed of sound, returned as a scalar or array, in m/s.

Data Types: `double`

P (Pa) — Air pressure
scalar | array

Air pressure, returned as a scalar or array, in Pa.

Data Types: `double`

ρ (kg/m³) — Air density
scalar | array

Air density, returned as scalar or array, in kg/m^3.

Data Types: `double`

ν (m²/s) — Kinematic viscosity
scalar | *m*-by-*n* array

Kinematic viscosity, returned as a scalar or *m*-by-*n* array, in m^2/s.

Dependencies

Selecting the **Output kinematic and dynamic viscosity** check box adds the **ν (m²/s)** output port.

Data Types: `double`

μ (N s/m²) — Dynamic viscosity
scalar | *m*-by-*n* array

Dynamic viscosity, returned as a scalar or *m*-by-*n* array, in $N\ s/m^2$.

Dependencies

Selecting the **Output kinematic and dynamic viscosity** check box adds the **μ (N s/m²)** output port.

Data Types: `double`

Parameters

S — Sutherland temperature
110.4 K (default) | scalar

Sutherland temperature in Kelvin, specified as a scalar.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** and **Change atmospheric parameters** are e.

T_{ref} — Reference temperature
273.15 K (default) | scalar

Reference temperature in Kelvin, specified as a scalar.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** is selected.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** and **Change atmospheric parameters** are e.

μ_{ref} — Reference dynamic viscosity
1.716e-5 kg/ms (default) | scalar

Reference dynamic viscosity in kilograms per meter second, specified as a scalar.

Dependencies

This optional input is available when the **Output kinematic and dynamic viscosity** and **Change atmospheric parameters** are e.

Output kinematic and dynamic viscosity — Option to output kinematic and dynamic viscosity
`off` (default) | `on`

Option to output kinematic and dynamic viscosity.

Dependencies

Selecting this check box:

- Adds the **ν (m^2/s)** output port.
- Adds the **μ (N s/m^2)** output port.

Dependencies

If selected simultaneously with the **Change atmospheric parameters** checkbox, adds the **Viscosity parameters** tab.

Programmatic Use

Block Parameter: `viscosityOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Change atmospheric parameters — Customize parameters
`off` (default) | `on`

Customize various atmospheric parameters to be different from the lapse rate values. Selecting this check box converts the block from Lapse Rate Model to ISA Atmosphere Model.

Dependencies

Selecting this check box enables the **Atmospheric parameters** tab with these parameters:

- **Acceleration due to gravity (m/s^2)**
- **Ratio of specific heats**
- **Characteristic gas constant (J/Kg/K)**
- **Lapse rate (K/m)**
- **Height of troposphere (m)**
- **Height of tropopause (m)**

- **Air density at mean sea level (Kg/m^3)**
- **Ambient pressure at mean sea level (N/m^2)**
- **Lowest altitude (m)**

Selecting in conjunction with **Output kinematic and dynamic viscosity** enables the **Viscosity parameters** tab with these parameters:

- **S**
- $\mathbf{T_{ref}}$
- $\boldsymbol{\mu_{ref}}$

Programmatic Use
Block Parameter: `custom`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Acceleration due to gravity (m/s^2) — Acceleration
`9.80665` (default) | scalar

Acceleration from gravity (*g*). in m/s^2, specified as double scalar.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `g`
Type: character vector
Values: double scalar
Default: `9.80665`

Ratio of specific heats — Ratio of heats
`1.4` (default) | scalar

Ratio of specific heats γ, specified as a double value.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `gamma`
Type: character vector
Values: double scalar
Default: `1.4`

Characteristic gas constant (J/Kg/K) — Gas constant
`287.0531` (default) | scalar

Characteristic gas constant (R), specified as double scalar, in J/Kg/K.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: R
Type: character vector
Values: double scalar
Default: 287.0531

Lapse rate (K/m) — Lapse rate
0.0065 (default) | scalar

Lapse rate of the troposphere, specified as double scalar, in K/m.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: L
Type: character vector
Values: double scalar
Default: 0.0065

Height of troposphere (m) — Troposphere height
11000 (default) | scalar

Height of the troposphere (range of decreasing temperatures), specified as double scalar, in m.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: h_trop
Type: character vector
Values: double scalar
Default: 11000

Height of tropopause (m) — Tropopause height
20000 (default) | scalar

Height of the tropopause (range of constant temperature), specified as double scalar, in m.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: h_strat
Type: character vector
Values: double scalar
Default: 20000

Air density at mean sea level (Kg/m^3) — Air density
1.225 (default) | scalar

Air density at mean sea level, specified as double scalar, in Kg/m^3.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `rho0`
Type: character vector
Values: double scalar
Default: `1.225`

Ambient pressure at mean sea level (N/m^2) — Ambient pressure
`101325` (default) | scalar

Ambient pressure at mean sea level, specified as double scalar, in N/m^2.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `P0`
Type: character vector
Values: double scalar
Default: `101325`

Ambient temperature at mean sea level (K) — Ambient temperature
`288.15` (default) | scalar

Ambient temperature at mean sea level (T_0), specified as double scalar, in K.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `K`
Type: character vector
Values: double scalar
Default: `101325`

Lowest altitude (m) — Lowest altitude
`0` (default) | scalar

Lowest altitude above which temperature and pressure lapse, specified as double scalar, in m. **Lowest altitude (m)** must be below **Height of tropopause**.

Dependencies

This parameter is enabled when the **Change atmospheric parameters** check box is selected.

Programmatic Use
Block Parameter: `h0`
Type: character vector
Values: double scalar
Default: `0`

Algorithms

These equations define the troposphere:

$$T = T_0 - Lh$$

$$P = P_0\left(\frac{T}{T_0}\right)^{\frac{g}{LR}}$$

$$\rho = \rho_0\left(\frac{T}{T_0}\right)^{\frac{g}{LR} - 1}$$

$$a = \sqrt{\gamma RT}$$

These equations define the tropopause (lower stratosphere):

$$T = T_0 - Lhts$$

$$P = P_0\left(\frac{T}{T_0}\right)^{\frac{g}{LR}} e^{\frac{g}{RT}(hts - h)}$$

$$\rho = \rho_0\left(\frac{T}{T_0}\right)^{\frac{g}{LR} - 1} e^{\frac{g}{RT}(hts - h)}$$

$$a = \sqrt{\gamma RT}$$

where:

T_0	Absolute temperature at mean sea level in kelvin (K)
ρ_0	Air density at mean sea level in kg/m ^{3}
P_0	Static pressure at mean sea level in N/m ^{2}
h	Altitude in m
hts	Height of the troposphere in m
T	Absolute temperature at altitude h in kelvin (K)
ρ	Air density at altitude h in kg/m ^{3}
P	Static pressure at altitude h in N/m ^{2}
a	Speed of sound at altitude h in m/s ^{2}
L	Lapse rate in K/m
R	Characteristic gas constant J/kg-K
γ	Specific heat ratio
g	Acceleration due to gravity in m/s ^{2}

Version History

Introduced before R2006a

References

[1] *U.S. Standard Atmosphere.*, Washington, D.C.: U.S. Government Printing Office, 1976.

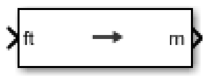
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
COESA Atmosphere Model | ISA Atmosphere Model

Length Conversion

Convert from length units to desired length units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Length Conversion block computes the conversion factor from specified input length units to specified output length units and applies the conversion factor to the input signal.

The Length Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Length
scalar | array

Length, specified as a scalar or array, in initial length units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Length
scalar | array

Length, returned as a scalar or array, in final length units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`ft` (default) | `m` | `km` | `in` | `mi` | `naut mi`

Input units, specified as:

m	Meters
ft	Feet
km	Kilometers
in	Inches
mi	Miles
naut mi	Nautical miles

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use

Block Parameter: IU
Type: character vector
Values: 'm' | 'ft' | 'km' | 'in' | 'mi' | 'naut mi'
Default: 'ft'

Final unit — Input units

m (default) | ft | km | in | mi | naut mi

Output units, specified as:

m	Meters
ft	Feet
km	Kilometers
in	Inches
mi	Miles
naut mi	Nautical miles

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use

Block Parameter: OU
Type: character vector
Values: 'm' | 'ft' | 'km' | 'in' | 'mi' | 'naut mi'
Default: 'm'

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

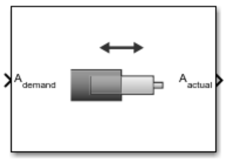
Generate C and C++ code using Simulink® Coder™.

See Also

Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

Linear Second-Order Actuator

Implement second-order linear actuator



Libraries:
Aerospace Blockset / Actuators

Description

The Second Order Linear Actuator block outputs the actual actuator position using the input demanded actuator position and other parameters that define the system.

Ports

Input

A_{demand} — Demanded actuator position
scalar | array

Demanded actuator position, specified as a scalar or array.

Data Types: `double`

Output

A_{actual} — Actual actuator position
scalar | array

Actual actuator position, returned as a scalar or array.

Data Types: `double`

Parameters

Natural frequency — Natural frequency

`1` (default) | scalar

Natural frequency of the actuator, specified as a scalar double, in radians per second.

Programmatic Use
Block Parameter: `wn_fin`
Type: character vector
Values: scalar | double
Default: `'1'`

Damping ratio — Damping ratio

`0.3` (default) | scalar

Damping ratio of the actuator, specified as a scalar double.

Programmatic Use
Block Parameter: `z_fin`
Type: character vector
Values: scalar | double
Default: `'0.3'`

Initial position — Initial position

`0` (default) | scalar

Initial position of the actuator, specified as a scalar double. The units of initial position must be the same as the $\mathbf{A_{demand}}$ input.

Programmatic Use
Block Parameter: `fin_act_0`
Type: character vector
Values: scalar | double
Default: `'0'`

Initial velocity — Initial velocity

`0` (default) | scalar

Initial velocity of the actuator, specified as a scalar double. The units of initial velocity must be the same as the $\mathbf{A_{demand}}$ input.

Programmatic Use
Block Parameter: `fin_act_vel`
Type: character vector
Values: scalar | double
Default: `'0'`

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Nonlinear Second-Order Actuator

Light Helicopter

Generic light helicopter

Description

Light Helicopter is one of the rotorcraft that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. For detailed views of the Light Helicopter, see "Views" on page 5-547.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Rotorcraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Parameters** tab, set the **Type** parameter to `Light Helicopter`.
3. Set the **Initial translation (m)** and **Initial rotation (rad)** parameters to an array size that matches the Light Helicopter rotorcraft, for example, `zeros(6, 3)`.

Data for Light Helicopter Placement

The `Light Helicopter` sample mesh origin is at ground level. To place the rotorcraft, consider using these values.

Airport Scene Placement

To place the `Light Helicopter` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, 0] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0, 0]`

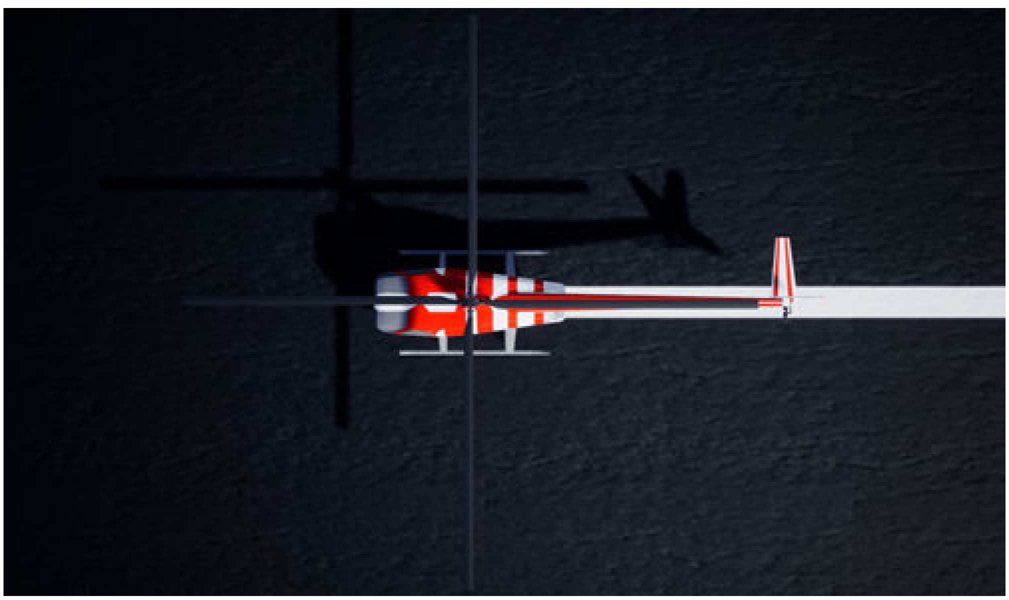
Altitude Sensor

For the altitude sensor in the Simulation 3D Rotorcraft block, use these values.

Parameter	Value
Body Z offset (m)	`1.1687`
Ground contact location 1 (m)	`[1.47, 1.04, 1.1687]`
Ground contact location 2 (m)	`[-1.47, 1.04, 1.1687]`
Ground contact location 3 (m)	`[-1.47, -1.04, 1.1687]`
Ground contact location 4 (m)	`[1.47, -1.04, 1.1687]`

Views

`Top-down view` — Light helicopter top-down view
diagram



`Side view` — Light helicopter side view
diagram

`Front view` — Light helicopter front view
diagram

`Back view` — Light helicopter back view
diagram

Lights and Skeleton

Lights

Light	Bone
Landing light	LandingLight
Nose light	NoseLight
Red navigation light	RedNavLight
Green navigation light	GreenNavLight
White navigation light	PositionLight
Strobe light	StrobeLight
Beacon light	BeaconLight

Skeleton

- LightHelicopter
  - `Engine`
  - `Rotor1`
  - `Rotor2`
  - `Sensor1`
  - `Sensor2`
  - `NoseLight`
  - `LandingLight`
  - `RedNavLight`
  - `GreenNavLight`
  - `BeaconLight`

- `StrobeLight`
- `PositionLight`

Version History

Introduced in R2023a

See Also

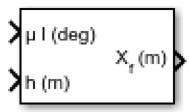
Simulation 3D Rotorcraft | Helicopter | Multirotor

Topics

"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

LLA to ECEF Position

Calculate Earth-centered Earth-fixed (ECEF) position from geodetic latitude, longitude, and altitude above planetary ellipsoid



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The LLA to ECEF Position block converts geodetic latitude ($\bar{\mu}$), longitude ($\bar{\iota}$), and altitude $\left(\bar{h}\right)$ above the planetary ellipsoid into a 3-by-1 vector of ECEF position ($\bar{p}$). Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles. For more information the ECEF position calculation, see "Algorithms" on page 5-553.

Limitations

- The planet is assumed to be ellipsoidal. To use a spherical planet, set the **Flattening** parameter to zero.
- The implementation of the ECEF coordinate system assumes that the origin is at the center of the planet, the *x*-axis intersects the Greenwich meridian and the equator, the *z*-axis is the mean spin axis of the planet, positive to the north, and the *y*-axis completes the right-handed system.

Ports

Input

μ l — Geodetic latitude and longitude
2-by-1 vector

Geodetic latitude and longitude, specified as a 2-by-1 vector, in degrees.

Data Types: `double`

h — Altitude
scalar

Altitude above the planetary ellipsoid, specified as a scalar.

Data Types: `double`

Output

X_f — Position
3-by-1 vector

Position in ECEF frame, returned as a 3-by-1 vector, in the same units as the input at the **h** port.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Parameter and output units:

Units	Radius from CG to Center of Planet	Equatorial Radius
`Metric (MKS)`	Meters	Meters
`English`	Feet	Feet

Dependencies

To enable this, set **Planet model** to `Earth (WGS84)`.

Programmatic Use

Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use

Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use

Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius of planet — Radius of planet at equator

6378137 (default) | scalar

Radius of the planet at its equator, in the same units as the desired units for ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use

Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `6378137`

Algorithms

The ECEF position is calculated from the geocentric latitude at mean sea-level (λ_s) and longitude using:

$$\bar{p} = \begin{bmatrix} \bar{p}_x \\ \bar{p}_y \\ \bar{p}_z \end{bmatrix} = \begin{bmatrix} r_s\cos\lambda_s\cos\imath + h\cos\mu\cos\imath \\ r_s\cos\lambda_s\sin\imath + h\cos\mu\sin\imath \\ r_s\sin\lambda_s + h\sin\mu \end{bmatrix},$$

where geocentric latitude at mean sea-level and the radius at a surface point (r_s) are defined by flattening $\left(\bar{f}\right)$, and equatorial radius $\left(\bar{R}\right)$ in the following relationships:

$$\lambda_s = \operatorname{atan}\left((1-f)^2\tan\mu\right)$$

$$r_s = \sqrt{\frac{R^2}{1 + \left(1/(1-f)^2 - 1\right)\sin^2\lambda_s}}$$

Version History

Introduced before R2006a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 1992.

[2] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. Second Edition. Reston, VA: AIAA Education Series, 2000.

[3] *Recommended Practice for Atmospheric and Space Flight Vehicle Coordinate Systems,* R-004-1992, ANSI/AIAA, February 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

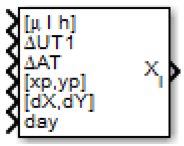
Direction Cosine Matrix ECEF to NED | Direction Cosine Matrix ECEF to NED to Latitude and Longitude | ECEF Position to LLA | Flat Earth to LLA | Radius at Geocentric Latitude

Topics

"About Aerospace Coordinate Systems" on page 2-8

LLA to ECI Position

Convert latitude, longitude, altitude (LLA) coordinates to Earth-centered inertial (ECI) coordinates



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The LLA to ECI Position block converts latitude, longitude, and altitude (LLA) coordinates to Earth-centered inertial (ECI) position coordinates, based on the specified reduction method and Coordinated Universal Time (UTC), for the specified time and geophysical data. The latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Ports

Input

[μ l h] — Latitude, longitude, and altitude
three-element vector

Latitude, longitude, and altitude values of coordinates to convert, specified as a three-element vector, in degrees.

Data Types: `double`

ΔUT1 — Difference between UTC and Universal Time
scalar

Difference between UTC and Universal Time (UT1) in seconds, specified as a scalar, for which the block calculates the direction cosine or transformation matrix.

Example: `0.234`

Dependencies

To enable this, select **Higher accuracy parameters**.

Data Types: `double`

ΔAT — Difference between International Atomic Time and UTC
scalar

Difference between International Atomic Time (IAT) and UTC, specified as a scalar, in seconds, for which the block calculates the direction cosine or transformation matrix.

Example: 32

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

[xp,yp] — Polar displacement of Earth
1-by-2 array

Polar displacement of Earth, specified as a 1-by-2 array, in radians, from the motion of the Earth crust, along the *x*- and *y*-axes.

Example: `[-0.0682e-5 0.1616e-5]`

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

Port_5 — Adjustment based on reduction method
1-by-2 array

Adjustment based on reduction method, specified as 1-by-2 array. The name of the port depends on the setting of the **Reduction** parameter:

- If reduction method is `IAU-2000/2006`, this input is the adjustment to the location of the Celestial Intermediate Pole (CIP), specified in radians. This location ([d*X*,d*Y*]) is along the *x*-axis and *y*-axis.
- If reduction method is `IAU-76/FK5`, this input is the adjustment to the longitude ([Δδψ, Δδε]), specified in radians.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/Products page.

Example: `[-0.2530e-6 -0.0188e-6]`

Dependencies

To enable this port, select **Higher accuracy parameters**.

Data Types: `double`

Port_6 — Time increment source
scalar

Time increment source, specified as a scalar, such as the Clock block.

Dependencies

- The port name and time increment depend on the **Time Increment** parameter.

Time Increment Value	Port Name
`Day`	**day**
`Hour`	**hour**
`Min`	**min**

Time Increment Value	Port Name
`Sec`	**sec**
`None`	No port

- To disable this port, set the **Time Increment** parameter to `None`.

Data Types: `double`

Output

X_i — Original position
3-by-1 element vector

Original position vector with respect to the ECI reference system, returned as a 3-by-1 element vector.

Data Types: `double`

Parameters

Reduction — Reduction method

`IAU-76/FK5` (default) | `IAU-2000/2006`

Reduction method to convert the coordinates. Method can be one of:

- `IAU-76/FK5`

  Reduce the calculation using the International Astronomical Union 76/Fifth Fundamental Catalogue (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, the block calculates the transformation matrix rather than the direction cosine matrix.

- `IAU-2000/2006`

  Reduce the calculation using the International Astronomical Union 2000/2006 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation.

Programmatic Use
Block Parameter: `red`
Type: character vector
Values: `'IAU-2000/2006'` | `'IAU-76/FK5'`
Default: `'IAU-2000/2006'`

Year — Year

`2013` (default) | double, whole number, greater than 1

Year to calculate the Coordinated Universal Time (UTC) date. Enter a double value that is a whole number greater than 1, such as `2013`.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: double, whole number, greater than 1
Default: `'2013'`

Month — Month

`January` (default) | `February` | `March` | `April` | `May` | `June` | `July` | `August` | `September` | `October` | `November` | `December`

Month to calculate the UTC date.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Day

`1` (default) | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10` | `11` | `12` | `13` | `14` | `15` | `16` | `17` | `18` | `19` | `20` | `21` | `22` | `23` | `24` | `25` | `26` | `27` | `28` | `29` | `30` | `31`

Day to calculate the UTC date.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'` | `'10'` | `'11'` | `'12'` | `'13'` | `'14'` | `'15'` | `'16'` | `'17'` | `'18'` | `'19'` | `'20'` | `'21'` | `'22'` | `'23'` | `'24'` | `'25'` | `'26'` | `'27'` | `'28'` | `'29'` | `'30'` | `'31'`
Default: `'1'`

Hour — Hour

`0` (default) | double, whole number, `0` to `24`

Hour to calculate the UTC date. Enter a double value that is a whole number, from `0` to `24`.

Programmatic Use
Block Parameter: `hour`
Type: character vector
Values: double, whole number, `0` to `24`
Default: `'0'`

Minutes — Minutes

`0` (default) | double, whole number, `0` to `60`

Minutes to calculate the UTC date. Enter a double value that is a whole number, from `0` to `60`.

Programmatic Use
Block Parameter: `min`
Type: character vector
Values: double, whole number, `0` to `60`
Default: `'0'`

Seconds — Seconds

`0` (default)

Seconds to calculate the UTC date. Enter a double value that is a whole number, from `0` to `60`.

Programmatic Use
Block Parameter: `sec`
Type: character vector
Values: double, whole number, `0` to `60`
Default: `'0'`

Time increment — Time increment

`Day` (default) | `None` | `Hour` | `Min` | `Sec`

Time increment between the specified date and the desired model simulation time. The block adjusts the calculated direction cosine matrix to take into account the time increment from model simulation. For example, selecting `Day` and connecting a simulation timer to the port means that each time increment unit is one day and the block adjusts its calculation based on that simulation time.

This parameter corresponds to the time increment input, the clock source.

If you select `None`, the calculated Julian date does not take into account the model simulation time.

Programmatic Use
Block Parameter: `deltaT`
Type: character vector
Values: `'None'` | `'Day'` | `'Hour'` | `'Min'` | `'Sec'`
Default: `'Day'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as follows.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `errorflag`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`

Higher accuracy parameters — Enable higher accuracy parameters

`on` (default) | `off`

Select this check box to allow the following as block inputs. These inputs let you better control the conversion result. See “Input” on page 5-555 for a description.

- $\Delta\ UT1$
- $\Delta\ AT$
- $[\ xp\ ,\ yp\ ]$
- $[\Delta\delta\psi,\ \Delta\delta\varepsilon]$ or $[\mathrm{d}\ X,\ \mathrm{d}\ Y\ ]$

Programmatic Use
Block Parameter: `extraparamflag`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Units — Units

`Metric (MKS)` (default) | `English`

Specifies the parameter and output units.

Units	Position	Equatorial Radius	Altitude
`Metric (MKS)`	Meters	Meters	Meters
`English`	Feet	Feet	Feet

Dependencies

To enable this parameter, set **Earth model** to `Earth (WGS84)`.

Programmatic Use
Block Parameter: `eunits`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Earth model — Earth model

`WGS84` (default) | `Custom`

Earth model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `earthmodel`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Earth model** to `Custom`.

Programmatic Use
Block Parameter: `flat`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator.

Dependencies

To enable this parameter, set **Earth model** to `Custom`.

Programmatic Use
Block Parameter: `eqradius`
Type: character vector
Values: double scalar
Default: `6378137`

Version History

Introduced in R2014a

Extended Capabilities

C/C++ Code Generation

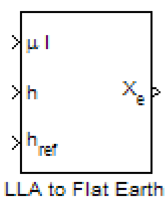
Generate C and C++ code using Simulink® Coder™.

See Also

ECI Position to LLA

LLA to Flat Earth

Estimate flat Earth position from geodetic latitude, longitude, and altitude



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The LLA to Flat Earth block converts a geodetic latitude ($\bar{\mu}$), longitude ($\bar{\iota}$), and altitude (h) into a 3-by-1 vector of flat Earth position ($\bar{p}$). Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles. For more information on the flat Earth coordinate system, see "Algorithms" on page 5-565.

Limitations

- This estimation method assumes that the flight path and bank angle are zero.
- This estimation method assumes the flat Earth z-axis is normal to the Earth at the initial geodetic latitude and longitude only. This method has higher accuracy over small distances from the initial geodetic latitude and longitude, and nearer to the equator. The longitude has higher accuracy with smaller variations in latitude. Additionally, longitude is singular at the poles.

Ports

Input

µl — Geodetic latitude and longitude
2-by-1 vector

Geodetic latitude and longitude, specified as a 2-by-1 vector, in degrees.

Data Types: `double`

h — Altitude
scalar

Altitude above the input reference altitude, specified as a scalar, in the same units as the flat Earth position.

Data Types: `double`

h_{ref} — Reference height
scalar

Reference height from the surface of the Earth to the flat Earth frame, specified as a scalar, in the same units as the flat Earth position. The reference height is estimated with regard to Earth frame.

Data Types: `double`

μ_{ref} l_{ref} — Reference location
2-by-1 vector

Reference location, specified as a 2-by-1 vector, in degrees of latitude and longitude, for the origin of the estimation and the origin of the flat Earth coordinate system. Use this port if you want to specify the reference location as a dynamic value.

Dependencies

To enable this port, select **Input reference position and orientation**.

Data Types: `double`

ψ_{ref} — Direction of flat Earth *x*-axis
scalar

Angle, specified as a scalar, for converting flat Earth *x* and *y* coordinates to North and East coordinates. Use this port if you want to specify the angle as a dynamic value.

Dependencies

To enable this port, select **Input reference position and orientation**.

Data Types: `double`

Output

X_e — Position
3-by-1 vector

Position in flat Earth frame, returned as a vector.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Parameter and output units:

Units	Position	Equatorial Radius	Altitude
`Metric (MKS)`	Meters	Meters	Meters
`English`	Feet	Feet	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Dependencies

Selecting the `Custom` option disables the **Units** parameter and enables these parameters:

- **Flattening**
- **Equatorial radius of planet**

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of Earth

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius of planet — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator, in the same units as the desired units for ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `6378137`

Input reference position and orientation — Input reference position and orientation as ports

`off` (default) | `on`

- To enable input ports for reference position and angle to convert flat Earth, select this check box.
- To specify the reference positions and angle as static values, clear this check box.

Select this check box if you want

Programmatic Use
Block Parameter: `refPosPort`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Reference geodetic latitude and longitude [deg] — Initial geodetic latitude and longitude

`[0 10]` (default) | 2-by-1 vector

Reference location in latitude and longitude, specified as 2-by-1 vector, in degrees.

Dependencies

To enable this parameter, clear **Input reference position and orientation**.

Programmatic Use
Block Parameter: `LL0`
Type: character vector
Values: 2-by-1 vector
Default: `[0 10]`

Direction of flat Earth x-axis (degrees clockwise from north) — Flat Earth *x* and *y* coordinates

`0` (default) | scalar

Angle to convert flat Earth *x* and *y* coordinates to North and East coordinates, specified as a scalar double, in degrees.

Dependencies

To enable this parameter, clear **Input reference position and orientation**.

Programmatic Use
Block Parameter: `psi`
Type: character vector
Values: double scalar
Default: `0`

Algorithms

The flat Earth coordinate system assumes the *z*-axis is downward positive. The estimation begins by finding the small changes in latitude and longitude from the output latitude and longitude minus the initial latitude and longitude.

$$d\mu = \mu - \mu_0$$
$$d\iota = \iota - \iota_0$$

To convert geodetic latitude and longitude to the North and East coordinates, the estimation uses the radius of curvature in the prime vertical (R_N) and the radius of curvature in the meridian (R_M). R_N and R_M are defined by the following relationships:

$$R_N = \frac{R}{\sqrt{1 - (2f - f^2)\sin^2\mu_0}}$$

$$R_M = R_N \frac{1 - (2f - f^2)}{1 - (2f - f^2)\sin^2\mu_0}$$

where (R) is the equatorial radius of the planet and f is the flattening of the planet.

Small changes in the North (dN) and East (dE) positions are approximated from small changes in the latitude and longitude by

$$dN = R_M\, d\mu$$
$$dE = R_N \cos\mu_0\, d\iota.$$

With the conversion of the North and East coordinates to the flat Earth x and y coordinates, the transformation has the form of

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} N \\ E \end{bmatrix},$$

where ψ is the angle in degrees clockwise between the x-axis and north.

The flat Earth z-axis value is the negative altitude minus the reference height (h_{ref}):

$$p_z = -h - h_{ref}.$$

Version History

Introduced in R2011a

References

[1] Stevens, B. L., and F. L. Lewis. *Aircraft Control and Simulation,* Hoboken, NJ: John Wiley & Sons, 2003.

[2] Etkin, B. *Dynamics of Atmospheric Flight* Hoboken, NJ: John Wiley & Sons, 1972.

Extended Capabilities

C/C++ Code Generation

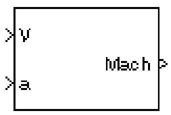
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix ECEF to NED | Direction Cosine Matrix ECEF to NED to Latitude and Longitude | ECEF Position to LLA | Flat Earth to LLA | Geocentric to Geodetic Latitude | LLA to ECEF Position | Radius at Geocentric Latitude

Mach Number

Compute Mach number using velocity and speed of sound



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Mach Number block computes the Mach number. The Mach number is defined as

$$Mach = \frac{\sqrt{V \cdot V}}{a},$$

where a is the speed of sound and V is the velocity vector.

Ports

Input

V — Velocity
3-element vector

Velocity vector, specified as an 3-element vector.

Data Types: `double`

a — Speed of sound
1-by-1 array

Speed of sound, specified as a 1-by-1 array.

Data Types: `double`

Output

Mach — Mach number
scalar

Mach number, returned as a scalar.

Data Types: `double`

Version History

Introduced before R2006a

Extended Capabilities

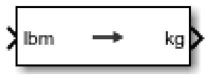
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Aerodynamic Forces and Moments | Dynamic Pressure

Mass Conversion

Convert from mass units to desired mass units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Mass Conversion block computes the conversion factor from specified input mass units to specified output mass units and applies the conversion factor to the input signal.

The Mass Conversion port block labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Mass
scalar | array

Mass, specified as a scalar or array, in initial mass units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Mass
scalar | array

Mass, returned as a scalar or array, in final mass units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`lbm` (default) | `kg` | `slug`

Input units, specified as.

`lbm`	Pound mass
`kg`	Kilograms
`slug`	Slugs

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use

Block Parameter: `IU`
Type: character vector
Values: `'lbm'` | `'kg'` | `'slug'`
Default: `'lbm'`

Final unit — Output units

`kg` (default) | `lbm` | `slug`

Output units, specified as:

`lbm`	Pound mass
`kg`	Kilograms
`slug`	Slugs

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use

Block Parameter: `OU`
Type: character vector
Values: `'lbm'` | `'kg'` | `'slug'`
Default: `'kg'`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

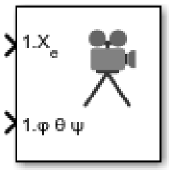
Generate C and C++ code using Simulink® Coder™.

See Also

Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Pressure Conversion | Temperature Conversion | Velocity Conversion

MATLAB Animation

Create six-degrees-of-freedom multibody custom geometry block



Libraries:
Aerospace Blockset / Animation / MATLAB-Based Animation

Description

The MATLAB Animation block creates a six-degrees-of-freedom multibody custom geometry block based on the `Aero.Animation` object. This block animates one or more vehicle geometries with *x*-*y*-*z* position and Euler angles through the specified bounding box, camera offset, and field of view. This block expects the rotation order *z*-*y*-*x* (psi, theta, phi).

To update the camera parameters in the animation, first set the parameters then close and double-click the block to reopen the MATLAB Animation window.

To access the parameters for this block, do one of:

- Right-click the block, then select **Mask** > **Mask Parameters**.
- Double-click the block to display the MATLAB Animation window, then click the **Block Parameters** icon.

Note The underlying graphics system stores values in single precision. As a result, you might notice that motion at coordinate positions greater than approximately 1e6 appear unstable. This is because a single-precision number has approximately six digits of precision. The instability is due to quantization at the local value of the `eps` MATLAB function. To visualize more stable motion for coordinates beyond 1e6, either offset the input data to a local zero, or scale down the coordinate values feeding the visualization.

Ports

Input

Port_1 — Downrange position, crossrange position, and altitude of vehicle
three-element vector

Downrange position, crossrange position, and altitude of the vehicle in Earth coordinates, specified as a three-element vector. The number on the port indicates the vehicle number.

Data Types: `double`

1.X_e — Downrange position, crossrange position, and altitude of vehicle
three-element vector

Downrange position, crossrange position, and altitude of the vehicle in Earth coordinates, specified as a three-element vector. The number on the port indicates the vehicle number.

Data Types: `double`

1.φ θ ψ — Euler angles
three-element vector

Euler angles (roll, pitch, and yaw) of the vehicle, specified as a three-element vector. The number on the port indicates the vehicle number.

Data Types: `double`

Port_N — Downrange position, crossrange position, and altitude (positive down)
three-element vector

Nth downrange position, crossrange position, and altitude (positive down) of the vehicle, specified as a three-element vector. The number on the port indicates the vehicle number.

Dependencies

To enable this port, select a **Vehicles** number from `2` to `10`.

Data Types: `double`

Port_N — Euler angles
three-element vector

Nth input Euler angles (roll, pitch, and yaw) of the vehicle, specified as a three-element vector. The number on the port indicates the vehicle number.

Dependencies

To enable this port, select a **Vehicles** number from `2` to `10`.

Data Types: `double`

Parameters

Vehicles — Vehicle to animate

`1` (default) | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10`

Vehicle to animate, specified from `1` to `10`.

Dependencies

Selecting a vehicle number from `2` to `10` adds corresponding input ports. Each vehicle has its own set of input ports, denoted by the number at the beginning of the input port label.

Programmatic Use
Block Parameter: `Vehicles`
Type: character vector
Values: `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10`
Default: `'1'`

Geometries (use 'quotes' on filenames) — Vehicle geometries `'astredwedge.mat'` (default) | MAT-file

Vehicle geometries, specified in a MAT-file. You can specify these geometries using:

- Variable name, for example `geomVar`
- Cell array of variable names, for example `{geomVar, AltGeomVar}`
- Character vector with single quotes, for example, `'astredwedge.mat'`
- Mixed cell array of variable names and character vectors, for example `{'file1.mat', 'file2.mat', 'file3.ac', geomVar}`

Note All specified geometries specified must exist in the MATLAB workspace and file names must exist in the current folder or be on the MATLAB path.

Programmatic Use
Block Parameter: `Geometries`
Type: character vector
Values: MAT-file
Default: `'astredwedge.mat'`

Bounding box coordinates — Boundary coordinates

`[-50,50,-50,50,-50,50]` (default) | six-element vector

Boundary coordinates for the vehicle, specified as a six-element vector.

This parameter is not tunable during simulation. A change to this parameter takes effect after simulation stops.

Programmatic Use
Block Parameter: `BoundingBoxCoordinates`
Type: character vector
Values: six-element vector
Default: `'[-50,50,-50,50,-50,50]'`

Camera offset — Distance from camera aim point to camera

`[-150,-50,0]` (default) | three-element vector

Distance from the camera aim point to the camera itself, specified as a three-element vector.

This parameter is not tunable during simulation. A change to this parameter takes effect after simulation stops.

Programmatic Use
Block Parameter: `CameraOffset`
Type: character vector
Values: three-element vector
Default: `'[-150,-50,0]'`

Camera view angle — Camera view angle

`3` (default) | scalar

Camera view angle, specified as a double scalar. By default, the camera aim point is the position of the first body lagged dynamically to indicate motion.

This parameter is not tunable during simulation. A change to this parameter takes effect after simulation stops.

Programmatic Use
Block Parameter: `CameraViewAngle`
Type: character vector
Values: double scalar
Default: `'3'`

Sample time — Sample time

`0.2` (default) | scalar

Sample time (-1 for inherited), specified as a double scalar.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: double scalar
Default: `'0.2'`

Version History

Introduced in R2007a

Extended Capabilities

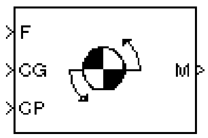
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

`Aero.Animation`

Moments about CG due to Forces

Compute moments about center of gravity due to forces applied at a point, not center of gravity



Libraries:
Aerospace Blockset / Mass Properties

Description

The Moments about CG due to Forces block computes moments about center of gravity due to forces that are applied at point CP, not at the center of gravity.

Ports

Input

F — Applied forces
3-element vector

Forces applied at point CP, specified as a three-element vector.

Data Types: `double`

CG — Center of gravity
3-element vector

Center of gravity, specified as three-element vector.

Data Types: `double`

CP — Application point of forces
3-element vector

Application point of forces, specified as a three-element vector.

Data Types: `double` | `bus`

Output

M — Moments at the center of gravity
3-element vector

Moments at the center of gravity in *x*-axis, *y*-axis and *z*-axis, returned as a three-element vector.

Data Types: `double`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

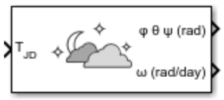
Generate C and C++ code using Simulink® Coder™.

See Also

Aerodynamic Forces and Moments | Estimate Center of Gravity

Moon Libration

Implement Moon librations



Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Description

The Moon Libration block implements the Moon librations using Chebyshev coefficients or a given Julian date. The block uses the Chebyshev coefficients that the NASA Jet Propulsion Laboratory provides.

Tip For T_{JD}, Julian date input for the block:

- Calculate the date using the Julian Date Conversion block or the Aerospace Toolbox `juliandate` function.
- Calculate the Julian date using some other means and input it using the Constant block.

Ports

Input

T_{JD} — Julian date
scalar | positive | between minimum and maximum Julian dates

Julian date, specified as a positive scalar between minimum and maximum Julian dates.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `Julian date`.

Data Types: `double`

$T0_{JD}$ — Fixed Julian date
scalar | positive

Fixed Julian date for a specific epoch that is the most recent midnight at or before the interpolation epoch, specified as a positive scalar. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian dates.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `T0 and elapsed Julian time`.

Data Types: `double`

ΔT_{JD} — Elapsed Julian time
scalar | positive

Elapsed Julian time between the fixed Julian date and the ephemeris time, specified as a positive scalar. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian date.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `T0 and elapsed Julian time`.

Data Types: `double`

Output

φ θ ψ (rad) — Euler angles
vector

Euler angles (φ θ ψ) for Moon attitude, in rad.

Data Types: `double`

ω (rad/day) — Moon libration Euler angular rate
vector

Moon libration Euler angular rates (ω), in rad/day.

Data Types: `double`

Parameters

Epoch — Epoch

`Julian date` (default) | `T0 and elapsed Julian time`

Epoch, specified as:

- `Julian date`

  Julian date to calculate the Moon libration. When this option is selected, the block has one input port, T_{JD}.
- `T0 and elapsed Julian time`

  Julian date, specified by two block inputs:

  - Fixed Julian date representing a starting epoch.
  - Elapsed Julian time between the fixed Julian date ($T0_{JD}$) and the desired model simulation time. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian dates.

Programmatic Use
Block Parameter: `epochflag`
Type: character vector
Values: `Julian date` | `T0 and elapsed Julian time`
Default: `'Julian date'`

Ephemeris model — Ephemeris model

DE405 (default) | DE421 | DE423 | DE430 | DE432t

Select one of these ephemerides models defined by the Jet Propulsion Laboratory. The block uses ephemeris data to calculate relative celestial positions required for third body point mass gravity and solar radiation pressure.

Ephemeris Model	Description
DE405	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE421	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE423	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE430	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE432t	Released in April 2014. This ephemeris takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. in the MATLAB desktop toolstrip, click **Add-Ons** .

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `ephemerisModel`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Action for out-of-range input — Out-of-range block behavior

None (default) | Warning | Error

Out-of-range block behavior, specified as follows.

Action	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: errorflag
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'Error'

Calculate rates — Calculate rate of Moon libration

on (default) | off

Select to calculate the rate of the Moon libration.

Dependencies

Select this check box to display the ω port.

Programmatic Use
Block Parameter: velflag
Type: character vector
Values: 'off' | 'on' |
Default: 'on'

Version History

Introduced in R2013a

References

[1] Folkner, W. M., J. G. Williams, D. H. Boggs. "The Planetary and Lunar Ephemeris DE 421." *IPN Progress Report* 42-178, 2009.

[2] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

Extended Capabilities

C/C++ Code Generation

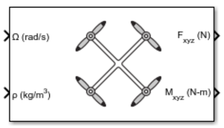
Generate C and C++ code using Simulink® Coder™.

See Also

aeroDataPackage | Earth Nutation | Planetary Ephemeris

Multirotor

Multipropeller dynamics



Libraries:
Aerospace Blockset / Propulsion

Description

The Multirotor block computes the aerodynamic forces and moments generated by multiple rotating propellers or rotors, such as quadcopters, in all three dimensions. You can also include simplified flap, or steady state, aerodynamics.

Limitations

- The block follows a simplified approach with the option to include steady state flap effects. It does not model dynamic flap, lag, or feathering motion of blade.
- The block does not directly consider the variation in pitch angle input, although you can perform approximate analysis by varying the blade twist.
- The block uses **Twist distribution** to model only linear or ideal twist distributions. The block assumes that the blade chord and lift curve slope are constant.
- The block does not include interference effects (between rotors) in the model.

Ports

Input

Ω (rad/sec) — Rotor speed
1-by-*N* vector

Rotor speed, specified as a 1-by-*N* vector in rad/sec in body frame.

Data Types: `double`

ρ — Air density
positive scalar

Air density, specified as a positive scalar in specified units.

Data Types: `double`

V_b (m/s) — Body velocity of rotor
3-by-1 vector | 1-by-3 vector

Body velocity of rotor, specified as a 3-by-1 or 1-by-3 vector, in body frame. To perform a multisystem analysis, consider connecting output from a State-Space or Integrator block to this port.

Data Types: `double`

ω_b — Angular velocity
3-by-1 vector | 1-by-3 vector

Angular velocity of entire vehicle, specified as a 3-by-1 or 1-by-3 vector in rad/s in body frame.

Data Types: `double`

Output

F_{xyz}(N) — Total force
three-element vector

Total force, returned as a three-element vector in body frame. Units depend on the **Units** parameter.

Data Types: `double`

M_{xyz}(Nm) — Net moment
three-element vector

Net moment in the *x*-*y*-*z* direction, returned as a three-element vector in body frame. Units depend on the **Units** parameter.

C_Q — Computed torque coefficient
scalar positive

Computed torque coefficient, returned as a scalar positive.

Dependencies

To enable this output port, select these check boxes:

- **Compute CT and CQ**
- **Output computed CT and CQ**

C_T — Computed thrust coefficient
scalar positive

Computed thrust coefficient, returned as a scalar positive.

Dependencies

To enable this output port, select these check boxes:

- **Compute CT and CQ**
- **Output computed CT and CQ**

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Units	Density	Force	Mass	Velocity	Radius	Chord	Hinge offset
Metric (MKS)	kg/m^3	Newtons	Newton-meter	Meters per second	Meters	Meters	Meters
English (Velocity in ft/s)	slug/ft^3	Pound force	Pound force-feet	Feet per second	Feet	Feet	Feet
English (Velocity in kts)	slug/ft^3	Pound force	Pound force-feet	Feet per second	Feet	Feet	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Modeling — Rotor thrust calculation method

`Without flap effects` (default) | `With flap effects`

Rotor thrust calculation method, specified as:

- `Without flap effects` — Model rotor thrust using force and moment calculations. For more information, see “Force and Moment” on page 5-590.
- `With flap effects` — Effect of tilt in rotor disc due to flap motion, while in forward flight, is included. The steady state lateral and longitudinal flap angles are calculated using the equations from [1] and [2].

Programmatic Use
Block Parameter: `modelMode`
Type: character vector
Values: `'Without flap effects'` | `'With flap effects'`
Default: `'Without flap effects'`

Vehicle

Configuration — Vehicle configuration

`Quadcopter X` (default) | `Quadcopter +`

Quadcopter configuration, specified as:

- `Quadcopter X` — Propellers configured as an X shape, considered as more stable than the `Quadcopter +` configuration.
- `Quadcopter +` — Propellers configured as a + shape.

For more information, see “Arm Computations” on page 5-588.

Programmatic Use
Block Parameter: `config`
Type: character vector
Values: `'Quadcopter X'` | `'Quadcopter +'`
Default: `'Quadcopter X'`

Arm length (m) — Vehicle arm length

`0.0624` (default) | nonzero positive scalar

Vehicle arm length, specified as a nonzero positive scalar. This length is the rotor displacement from the center of mass of the vehicle. For more information on arm lengths, see “Algorithms” on page 5-588.

Programmatic Use
Block Parameter: `l`
Type: character vector
Values: `'0.0624'` | nonzero positive scalar
Default: `'0.0624'`

Height (m) — Vertical offset from vehicle center of gravity

`-0.0159` (default) | nonzero scalar

Vertical offset from vehicle center of gravity, specified as a nonzero scalar.

Programmatic Use
Block Parameter: `h`
Type: character vector
Values: `'-0.0159'` | nonzero positive scalar
Default: `'-0.0159'`

Rotor

Number of blades — Number of blades per rotor

2 (default) | nonzero positive scalar

Number of blades per rotor, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `Nb`
Type: character vector
Values: `'2'` | nonzero positive scalar
Default: `'2'`

Compute CT and CQ — Option to calculate thrust coefficient and torque coefficient

on (default) | off

Select this check box to let the block calculate the thrust coefficient and torque coefficient. For more information on these calculations, see “Thrust Coefficient and Torque Coefficient Calculations” on page 5-590. The block assumes the aerodynamic and structural parameters to be constant.

Clear this check box to specify thrust coefficient and torque coefficient values.

Programmatic Use
Block Parameter: `CTcheck`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Output computed CT and CQ — Output computed thrust coefficient and torque coefficient

on (default) | off

Select this check box to output the calculated thrust coefficient and torque coefficient to $\mathbf{C_T}$ and $\mathbf{C_Q}$ output ports. For more information on these calculations, see “Thrust Coefficient and Torque Coefficient Calculations” on page 5-590. The block assumes the aerodynamic and structural parameters to be constant.

Otherwise, clear this check box.

Programmatic Use
Block Parameter: `CTout`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Thrust coefficient (CT) — Thrust coefficient

`0.0107` (default) | nonzero positive scalar

Thrust coefficient, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, clear the **Compute CT and CQ** check box.

Programmatic Use
Block Parameter: `CT`
Type: character vector
Values: `'0.0107'` | nonzero positive scalar
Default: `'0.0107'`

Torque coefficient (CQ) — Torque coefficient

`7.8263e-4` (default) | nonzero positive scalar

Torque coefficient, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, clear the **Compute CT and CQ** check box.

Programmatic Use
Block Parameter: `CQ`
Type: character vector
Values: `'7.8263e-4'` | nonzero positive scalar
Default: `'7.8263e-4'`

Radius (m) — Rotor radius

`0.0330` (default) | nonzero positive scalar

Rotor radius, specified as a nonzero positive scalar.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: `'0.0330'` | nonzero positive scalar
Default: `'0.0330'`

Chord (m) — Blade chord

`0.0080` (default) | nonzero positive scalar

Blade chord, specified as a nonzero positive scalar.

Programmatic Use
Block Parameter: `c`
Type: character vector
Values: `'0.0080'` | nonzero positive scalar
Default: `'0.0080'`

Hinge offset (m) — Hinge offset

`0` (default) | positive scalar

Hinge offset, specified as a positive scalar. This value is typically `0` for propellers.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: `'0'` | positive scalar
Default: `'0'`

Lift curve slope (per rad) — Lift curve slope

`5.5` (default) | nonzero positive scalar

Lift curve slope, specified as a nonzero positive scalar. The block assumes the aerodynamic and structural parameters to be constant. The block does not consider variation with respect to angle of attack.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `clalpha`
Type: character vector
Values: `'5.5'` | nonzero positive scalar
Default: `'5.5'`

Lock number — Lock number

`0.6051` (default) | nonzero positive scalar

Lock number, which is the ratio of aerodynamics forces to inertial forces, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `gamma`
Type: character vector
Values: `'0.6051'` | nonzero positive scalar
Default: `'0.6051'`

Twist distribution — Rotor blade twist distribution

`Linear` (default) | `Ideal`

Rotor blade twist distribution, specified as:

- `Linear` — Close approximation of blade twist distribution.
- `Ideal` — Optimal approximation of blade twist distribution.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `twistType`
Type: character vector
Values: `'Linear'` | `'Ideal'`
Default: `'Linear'`

Blade root angle (rad) — Blade root pitch angle

`0.2548` (default) | nonzero positive scalar

Blade root pitch angle θ_0, specified as a nonzero positive scalar.

Dependencies

To enable this parameter:

- Set **Modeling** to `With flap effects`.
- Set **Twist distribution** to `Linear`.

Programmatic Use
Block Parameter: `theta0`
Type: character vector
Values: `'0.2548'` | nonzero positive scalar
Default: `'0.2548'`

Blade twist angle (rad) — Blade twist angle

`-0.1361` (default) | positive scalar

Blade twist angle θ_1, specified as a positive scalar.

Dependencies

To enable this parameter:

- Set **Modeling** to `With flap effects`.
- Set **Twist distribution** to `Linear`.

Programmatic Use
Block Parameter: `theta1`
Type: character vector
Values: `'-0.1361'` | positive scalar
Default: `'-0.1361'`

Blade tip angle (rad) — Blade tip pitch angle

`0.06` (default) | nonzero positive scalar

Blade tip pitch angle θ_{tip} for ideal twist distribution, specified as a nonzero positive scalar.

Dependencies

To enable this parameter:

- Set **Modeling** to `With flap effects`.
- Set **Twist distribution** to `Ideal`.

Programmatic Use
Block Parameter: `thetaTip`
Type: character vector
Values: `'0.06'` | nonzero positive scalar
Default: `'0.06'`

Algorithms

Arm Computations

Quadcopter x

$p1=(d,-d,-h)$

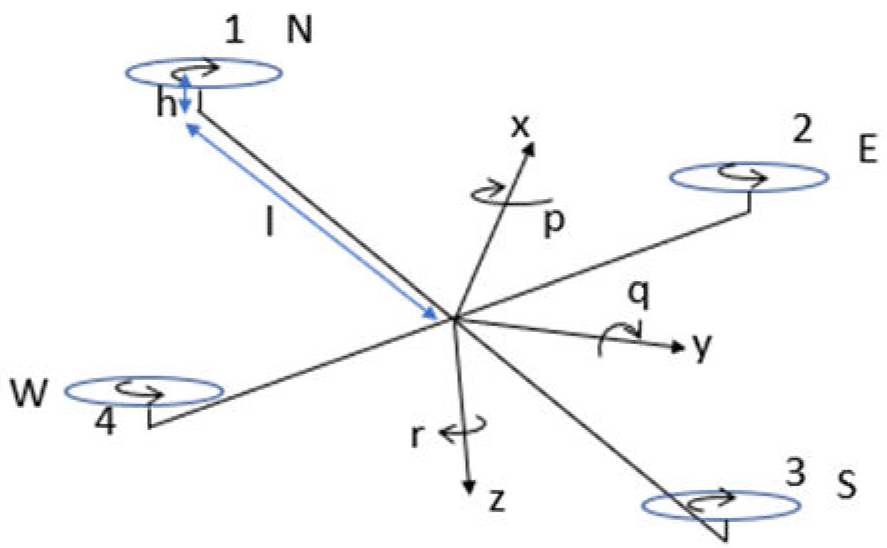
$p2=(d,d,-h)$

$p3=(-d,d,-h)$

$p4=(-d,-d,-h)$

where:

- l is the arm length.
- h is the offset from the vehicle center of gravity.



In this graphic:

- The x-, y-, z-axes are perpendicular to each other.
- The four arms of the vehicle are perpendicular and equal in length (l).
- All propellers are at same height (h).

Quadcopter +

$p1=(l,0,h)$

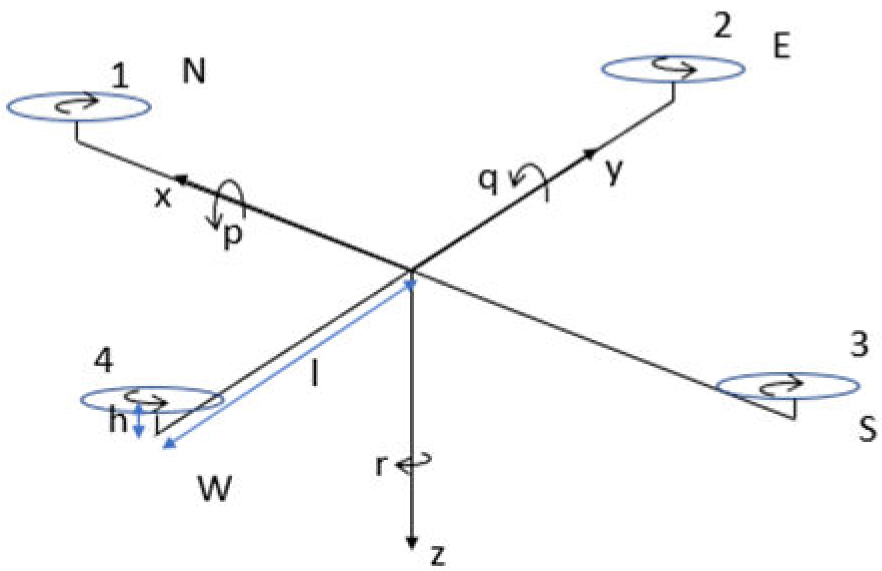
$p2=(0,l,h)$

$p3=(-l,0,h)$

$p4=(0,-l,h)$

where:

- l is the arm length.
- h is the offset from the vehicle center of gravity.



In this graphic:

- The x-, y-, z-axes are perpendicular to each other.
- The four arms of the vehicle are perpendicular and equal in length (l).
- All propellers are at same height (h).

Force and Moment

Force and moment calculated using these equations:

$$F\_xyz = \left[0;\ 0;\ -b\omega^2\right]$$
$$M\_xyz = \left[0;\ 0;\ k\omega^2\right]$$

where:

- ω is the rotor in revolutions per minute (RPM).
- b and k are experimental values:

  $$b = C_T\rho(\pi R^2)R^2$$
  $$k = -C_Q\rho(\pi R^2)R^3$$

  where:

  - C_T is the thrust coefficient
  - C_Q is the torque coefficient

Thrust Coefficient and Torque Coefficient Calculations

When the Compute CT and CQ check box is selected, the block calculates the thrust coefficient and torque coefficient using these equations.

With the inclusion of Prandtl's tip loss function, the incremental thrust coefficient using blade element momentum theory is:

$$dC_T = 4F\lambda^2 r\,dr$$

where F is the correction factor:

$$F = \frac{2}{\pi}\cos^{-1} exp - f$$

$$f = \frac{1}{2}\frac{Nb(1-r)}{r\varphi}$$

$$\varphi = \frac{\lambda}{r}$$

Using blade element theory, the incremental thrust coefficient is:

$$dC_T = \frac{1}{2}\sigma c_{l\alpha}(\theta(r)r - \lambda)rdr$$

with ideal twist,

$$\theta(r) = \frac{\theta_t}{r}$$

Resulting in:

$$dC_T = .5\sigma c_{l\alpha}(\theta_{t-\lambda r})rdr$$

Equating the two equations for dC_T, we have:

$$4F\lambda^2 = .5\sigma c_{l\alpha}(\theta_t - \lambda)$$

The equation can be solved to obtain λ using the `fsolve` from the Optimization Toolbox. To remove the dependency on the toolbox, the Newton-Raphson method is used:

$$fun = @(\lambda)4F(r,\lambda)\lambda^2 - .5\sigma c_{l\alpha}(\theta_t - \lambda)$$

The differential of *fun* as `diff`(*fun*,λ):

$$dfun = \frac{c_{l\alpha}\sigma}{2} + \frac{5734161139222659{*}\lambda{*}\cos^{-1}(\exp\left(\frac{Nb*(r-1)}{2*\lambda}\right)}{1125899906842624}$$
$$+\frac{5734161139222659{*}\mathrm{N}_b * \exp\left(\frac{Nb*(r-1)}{2*\lambda}\right) * (r-1)}{(4503599627370496{*}(1\text{-}\exp\left(\frac{Nb*(r-1)}{\lambda}\right))^{\left(\frac{1}{2}\right)})}$$

To avoid numerical issues, the ratios are approximated as:

$$dfun = \frac{c_{l\alpha}}{2} + 5.0930 * \lambda * \cos^{-1}\left(\exp\left(\frac{Nb*(r-1)}{2*\lambda}\right)\right)$$
$$+\frac{1.2732 * Nb * \exp\left(\frac{Nb*(r-1)}{2*\lambda} * (r-1)\right)}{(1-\exp\left(\frac{Nb*(r-1)}{\lambda}\right)^{\left(\frac{1}{2}\right)})}$$

After λ is calculated, C_T, C_Q are calculated by summing up $dC_T=4\lambda^2 r\,dr$ and $dC_Q=\lambda dC_T$ across the radius.

Version History

Introduced in R2023a

References

[1] Pounds, P. E. I. (2007). *Design, construction and control of a large quadrotor micro air vehicle* (Doctoral dissertation, Australian National University).

[2] Riether, F. (2016). *Agile quadrotor maneuvering using tensor-decomposition-based globally optimal control and onboard visual-inertial estimation* (Doctoral dissertation, Massachusetts Institute of Technology).

See Also

Rotor

Multirotor

Generic multirotor

Description

Multirotor is one of the rotorcraft that you can use within the 3D simulation environment. This topic describes two provided sample multirotor rotorcraft, Quadcopter and Mini Quadcopter. This environment is rendered using the Unreal Engine from Epic Games. This topic provides detailed views for two multirotors, a quadcopter and a mini quadcopter. For more information, see "Views" on page 5-595.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Rotorcraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Parameters** tab, set the **Type** parameter to `Multirotor`.
3. Set the **Initial translation (m)** and **Initial rotation (rad)** parameters to an array size that matches Multirotor rotorcraft, for example, `zeros(11, 3)`.

Data for Multirotor Placement for Quadcopter

The `Quadcopter` sample mesh origin is at ground level. To correctly place the rotorcraft, consider using these values.

Airport Scene Placement

To place the `Quadcopter` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, 0] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0, 0]`

Altitude Sensor

For the altitude sensor in the Simulation 3D Rotorcraft block, use these values.

Parameter	Value
Body Z offset (m)	`0.0934`
Ground contact location 1 (m)	`[0.1833, 0.1833, 0.0934]`
Ground contact location 2 (m)	`[-0.1833, 0.1833, 0.0934]`
Ground contact location 3 (m)	`[-0.1833, -0.1833, 0.0934]`
Ground contact location 4 (m)	`[0.1833, -0.1833, 0.0934]`

Data for Multirotor Placement for Mini Quadcopter

The `Mini Quadcopter` sample mesh origin is at ground level. To correctly place the rotorcraft, consider using these values.

Airport Scene Placement

To place the `Mini Quadcopter` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, 0] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0, 0]`

Altitude Sensor

For the altitude sensor in the Simulation 3D Rotorcraft block, use these values.

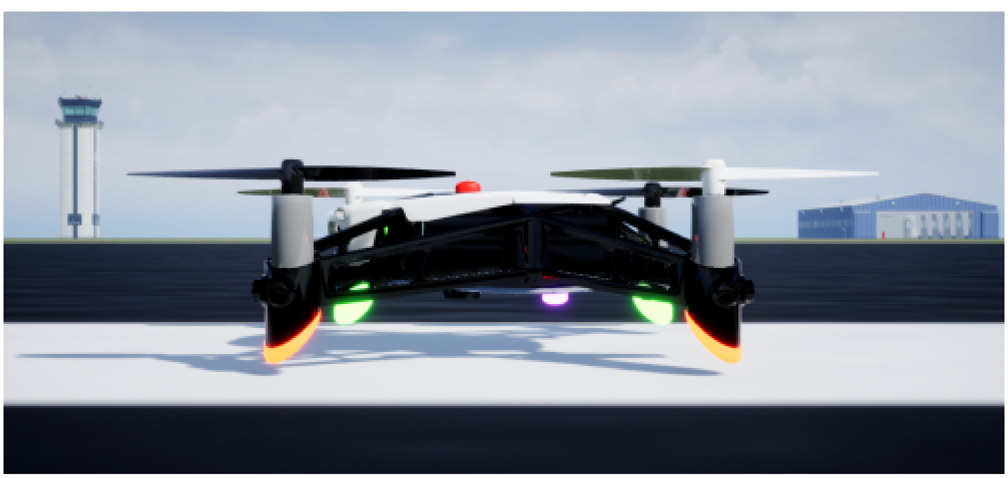
Parameter	Value
Body Z offset (m)	`0.0889`
Ground contact location 1 (m)	`[0.042, 0.042, 0.0889]`
Ground contact location 2 (m)	`[-0.042, 0.042, 0.0889]`
Ground contact location 3 (m)	`[-0.042, -0.042, 0.0889]`
Ground contact location 4 (m)	`[0.042, -0.042, 0.0889]`

Views

Quadcopter

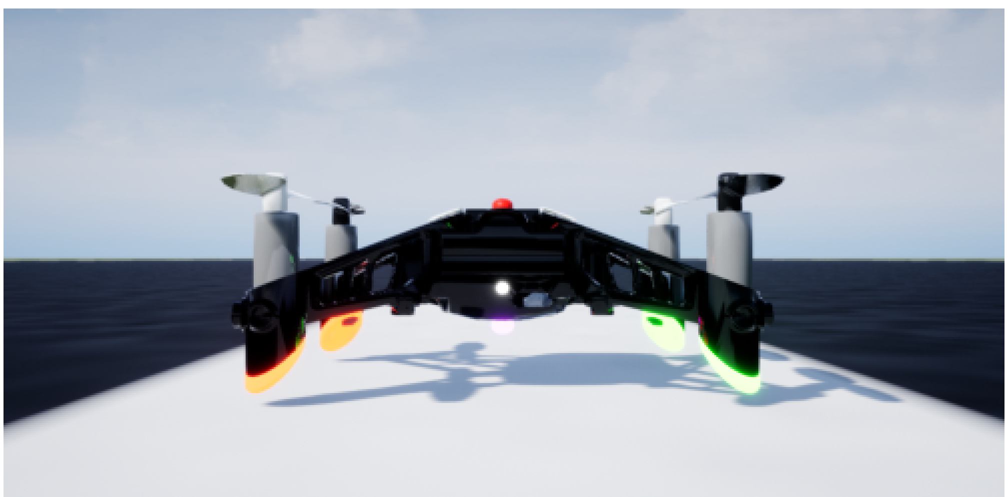
`Top-down view` — Quadcopter top-down view
diagram

`Side view` — Quadcopter side view
diagram



`Front view` — Quadcopter front view
diagram

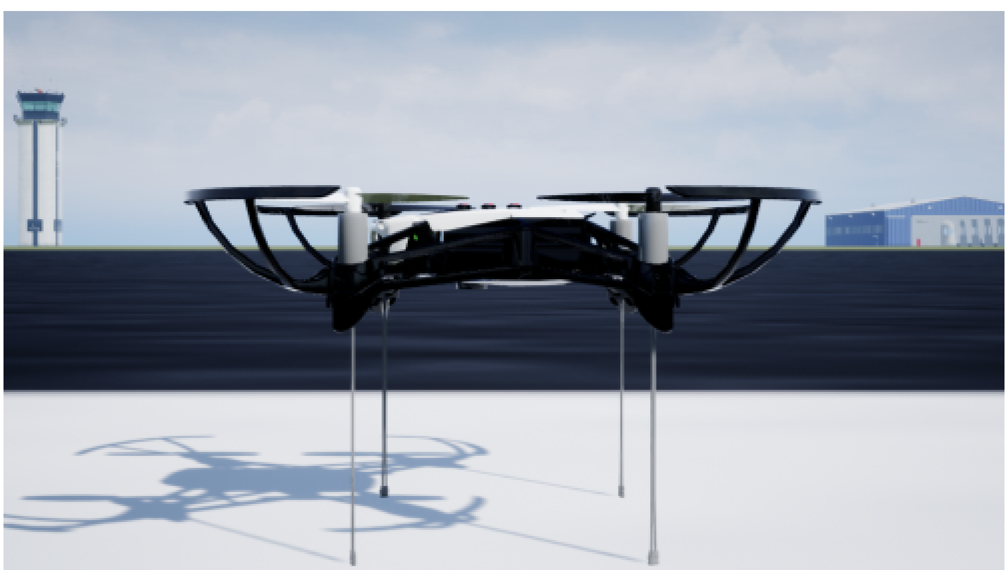
`Back view` — Quadcopter back view
diagram



Mini Quadcopter

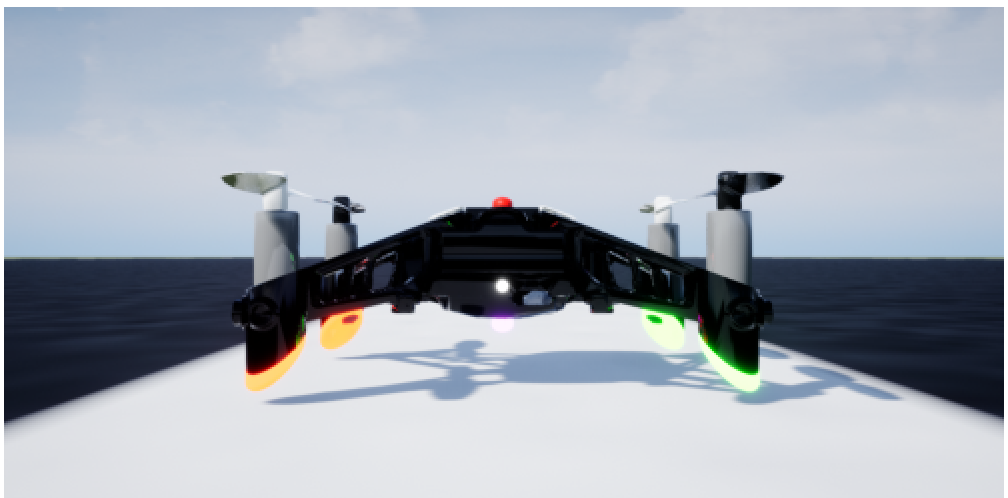
`Top-down view` — Quadcopter top-down view
diagram

`Side view` — Mini quadcopter side view
diagram



`Front view` — Mini quadcopter front view
diagram

`Back view` — Mini quadcopter back view
diagram



Lights and Skeleton

Lights

Light	Bone
Landing light	`LandingLight`
Nose light	`NoseLight`
Red navigation light	`RedNavLight`

Light	Bone
Green navigation light	`GreenNavLight`
White navigation light	`PositionLight`
Strobe light	`StrobeLight`
Beacon light	`BeaconLight`

Skeleton

- `Multirotor`
  - `Rotor1`
  - `Rotor2`
  - `Rotor3`
  - `Rotor4`
  - `Rotor5`
  - `Rotor6`
  - `Rotor7`
  - `Rotor8`
  - `Sensor1`
  - `Sensor2`
  - `NoseLight`
  - `LandingLight`
  - `RedNavLight`
  - `GreenNavLight`
  - `BeaconLight`
  - `StrobeLight`
  - `PositionLight`

Version History

Introduced in R2023a

See Also

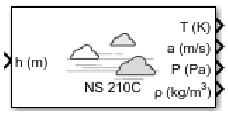
Simulation 3D Rotorcraft | Helicopter | Light Helicopter

Topics

"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Non-Standard Day 210C

Implement MIL-STD-210C climatic data



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The **Non-Standard Day 210C** block implements a portion of the climatic data of the MIL-STD-210C worldwide air environment to 80 km (geometric or approximately 262,467 feet geometric) for absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.

The COESA Atmosphere Model, Non-Standard Day 210C, and Non-Standard Day 310 blocks are identical blocks. When configured for COESA Atmosphere Model, the block implements the COESA mathematical representation. When configured for Non-Standard Day 210C, the block implements MIL-STD-210C climatic data. When configured for Non-Standard Day 310, the block implements MIL-HDBK-310 climatic data.

The COESA Atmosphere Model block port labels change based on the input and output units selected from the **Units** list.

Limitations

All values are held below the geometric altitude of 0 m (0 feet) and above the geometric altitude of 80,000 meters (approximately 262,467 feet). The envelope atmospheric model has a few exceptions where values are held below the geometric altitude of 1 kilometer (approximately 3,281 feet) and above the geometric altitude of 30,000 meters (approximately 98,425 feet). These exceptions arise from lack of data in MIL-STD-210C for these conditions.

In general, temperature values are interpolated linearly, and density values are interpolated logarithmically. Pressure and speed of sound are calculated using a perfect gas law. The envelope atmospheric model has a few exceptions where the extreme value is the only value provided as an output. Pressure in these cases is interpolated logarithmically. These envelope atmospheric model exceptions apply to all cases of high and low pressure, high and low temperature, and high and low density, excluding the extreme values and 1% frequency of occurrence. These exceptions arise from lack of data in MIL-STD-210C for these conditions.

Another limitation is that climatic data for the region south of 60°S latitude is excluded from consideration in MIL-STD-210C.

This block uses the metric version of data from the MIL-STD-210C specifications. Certain data within the envelope are inconsistent between metric and English versions for low density, low temperature, high temperature, low pressure, and high pressure. The most significant differences occur in the following values:

- For low density envelope data with 5% frequency, the density values in metric units are inconsistent at 4 km and 18 km and the density values in English units are inconsistent at 14 km.

- For low density envelope data with 10% frequency,
  - The density values in metric units are inconsistent at 18 km.
  - The density values in English units are inconsistent at 14 km.
- For low density envelope data with 20% frequency, the density values in English units are inconsistent at 14 km.
- For low temperature envelope data with 20% frequency, the temperature values at 20 km are inconsistent.
- For high pressure envelope data with 10% frequency, the pressure values in metric units at 8 km are inconsistent.

Ports

Input

Port_1 — Geopotential height
scalar | array

Geopotential height, specified as a scalar or array, in specified units.

Data Types: `double`

Output

Port_2 — Temperature
scalar | array

Temperature, specified as a scalar or array, in specified units.

Data Types: `double`

Port_2 — Speed of sound
scalar | array

Speed of sound, specified as a scalar or array, in specified units.

Data Types: `double`

Port_3 — Air pressure
scalar | array

Air pressure, specified as a scalar or array, in specified units.

Data Types: `double`

Port_4 — Air density
scalar | array

Air density, specified as a scalar or array, in specified units.

Data Types: `double`

Parameters

Units — Input and output units
Metric (MKS) (default) | English (Velocity in ft/s) | English (Velocity in kts)

Input and output units, specified as:

Units	Height	Temperature	Speed of Sound	Air Pressure	Air Density
Metric (MKS)	Meters	Kelvin	Meters per second	Pascal	Kilograms per cubic meter
English (Velocity in ft/s)	Feet	Degrees Rankine	Feet per second	Pound-force per square inch	Slug per cubic foot
English (Velocity in kts)	Feet	Degrees Rankine	Knots	Pound-force per square inch	Slug per cubic foot

Programmatic Use
Block Parameter: units
Type: character vector
Values: 'Metric (MKS)' | 'English (Velocity in ft/s)' | 'English (Velocity in kts)'
Default: 'Metric (MKS)'

Specification — Atmosphere model type
1976 COESA-extended U.S. Standard Atmosphere (default) | MIL-HDBK-310 | MIL-STD-210C

Atmosphere model type, specified as 1976 COESA-extended U.S. Standard Atmosphere, MIL-HDBK-310, or MIL-STD-210C. For the MIL-HDBK-310 and MIL-STD-210C options:

MIL-HDBK-310	This selection is linked to the Non-Standard Day 310 block. See the block reference for more information. Selecting MIL-HDBK-310 enables the parameters **Atmospheric model type**, **Extreme parameter**, **Frequency of occurrence**, and **Altitude of extreme value**.
MIL-STD-210C	This selection is linked to the Non-Standard Day 210C block. See the block reference for more information. Selecting MIL-HDBK-310 enables the parameters **Atmospheric model type**, **Extreme parameter**, **Frequency of occurrence**, and **Altitude of extreme value**.

Dependencies

Selecting MIL-HDBK-310 or MIL-STD-210C enables these parameters:

- **Atmospheric model type**
- **Extreme parameter**
- **Frequency of occurrence**

- **Altitude of extreme value**

Programmatic Use
Block Parameter: `spec`
Type: character vector
Values: `'1976 COESA-extended U.S. Standard Atmosphere'` | `'MIL-HDBK-310'` | `'MIL-STD-210C'`
Default: `'1976 COESA-extended U.S. Standard Atmosphere'`

Atmospheric model type — Model type
`Profile` (default) | `Envelope`

Representation of atmospheric model type, specified as:

`Profile`	Realistic atmospheric profiles associated with extremes at specified altitudes. Recommended for simulation of vehicles vertically traversing the atmosphere or when the total influence of the atmosphere is needed.
`Envelope`	Uses extreme atmospheric values at each altitude. Recommended for vehicles only horizontally traversing the atmosphere without much change in altitude.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- Selecting `Profile` enables the **Attitude of extreme value** parameter.

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'Profile'` | `'Envelope'`
Default: `'Profile'`

Extreme parameter — Model type
`High temperature` (default) | `Low temperature` | `High density` | `Low density` | `High pressure` | `Low pressure`

Atmospheric parameter that is the extreme value.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- The `High pressure` and `Low pressure` options appear only when **Atmospheric model type** is set to `Envelope`.

Programmatic Use
Block Parameter: `profile_var`
Type: character vector
Values: `'High temperature'` | `'Low temperature'` | `'High density'` | `'Low density'` | `'High pressure'` | `'Low pressure'`
Default: `'High temperature'`

Frequency of occurrence — Model type
`1%` (default) | `Extreme values` | `5%` | `10%` | `20%`

Percent of time the values would occur.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- `Extreme values`, `5%`, and `20%` are available only when `Envelope` is selected for **Atmospheric model type**.
- `1%` and `10%` are always available.

Programmatic Use
Block Parameter: `profile_percent`
Type: character vector
Values: `'Extreme values'` | `'1%'` | `'5%'` | `'10%'` | `'20%'`
Default: `'1%'`

Altitude of extreme value — Geometric altitude
`5 km (16404 ft)` (default) | `10 km (32808 ft)` | `20 km (65617 ft)` | `30 km (98425 ft)` | `40 km (131234 ft)`

Geometric altitude at which the extreme values occur, specified as `5 km (16404 ft)`, `10 km (32808 ft)`, `20 km (65617 ft)`, `30 km (98425 ft)`, or `40 km (131234 ft)`.

Dependencies

This parameter appears if the **Atmospheric model type** is set to `Profile`.

Programmatic Use
Block Parameter: `profile_alt`
Type: character vector
Values: `5 km (16404 ft)` | `10 km (32808 ft)` | `20 km (65617 ft)` | `30 km (98425 ft)` | `40 km (131234 ft)`
Default: `40 km (131234 ft)`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced before R2006a

References

[1] *Global Climatic Data for Developing Military Products.* MIL-STD-210C, Washington, D.C.: Department of Defense, 1987.

Extended Capabilities

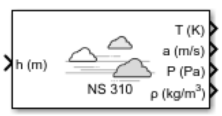
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

CIRA-86 Atmosphere Model | COESA Atmosphere Model | ISA Atmosphere Model | Non-Standard Day 310

Non-Standard Day 310

Implement MIL-HDBK-310 climatic data



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The Non-Standard Day 310 block implements a portion of the climatic data of the MIL-HDBK-310 worldwide air environment to 80 km (geometric or approximately 262,467 feet geometric) for absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.

The COESA Atmosphere Model, Non-Standard Day 210C, and Non-Standard Day 310 blocks are identical blocks. When configured for COESA Atmosphere Model, the block implements the COESA mathematical representation. When configured for Non-Standard Day 210C, the block implements MIL-STD-210C climatic data. When configured for Non-Standard Day 310, the block implements MIL-HDBK-310 climatic data.

The COESA Atmosphere Model block port labels change based on the input and output units selected from the **Units** list.

Limitations

All values are held below the geometric altitude of 0 m (0 feet) and above the geometric altitude of 80,000 meters (approximately 262,467 feet). The envelope atmospheric model has a few exceptions where values are held below the geometric altitude of 1 kilometer (approximately 3,281 feet) and above the geometric altitude of 30,000 meters (approximately 98,425 feet). These exceptions arise from lack of data in MIL-HDBK-310 for these conditions.

In general, temperature values are interpolated linearly, and density values are interpolated logarithmically. Pressure and speed of sound are calculated using a perfect gas law. The envelope atmospheric model has a few exceptions where the extreme value is the only value provided as an output. Pressure in these cases is interpolated logarithmically. These envelope atmospheric model exceptions apply to all cases of high and low pressure, high and low temperature, and high and low density, excluding the extreme values and 1% frequency of occurrence. These exceptions arise from lack of data in MIL-HDBK-310 for these conditions.

Another limitation is that climatic data for the region south of 60°S latitude is excluded from consideration in MIL-HDBK-310.

This block uses the metric version of data from the MIL-STD-310 specifications. Certain data within the envelope are inconsistent between metric and English versions for low density, low temperature, high temperature, low pressure, and high pressure. The most significant differences occur in the following values:

- For low density envelope data with 5% frequency, the density values in metric units are inconsistent at 4 km and 18 km and the density values in English units are inconsistent at 14 km.

- For low density envelope data with 10% frequency,
  - The density values in metric units are inconsistent at 18 km.
  - The density values in English units are inconsistent at 14 km.
- For low density envelope data with 20% frequency, the density values in English units are inconsistent at 14 km.
- For low temperature envelope data with 20% frequency, the temperature values at 20 km are inconsistent.
- For high pressure envelope data with 10% frequency, the pressure values in metric units at 8 km are inconsistent.

Ports

Input

Port_1 — Geopotential height
scalar | array

Geopotential height, specified as a scalar or array, in specified units.

Data Types: `double`

Output

Port_1 — Temperature
scalar | array

Temperature, specified as a scalar or array, in specified units.

Data Types: `double`

Port_2 — Speed of sound
scalar | array

Speed of sound, specified as a scalar or array, in specified units.

Data Types: `double`

Port_3 — Air pressure
scalar | array

Air pressure, specified as a scalar or array, in specified units.

Data Types: `double`

Port_4 — Air density
scalar | array

Air density, specified as a scalar or array, in specified units.

Data Types: `double`

Parameters

Units — Input and output units
Metric (MKS) (default) | English (Velocity in ft/s) | English (Velocity in kts)

Input and output units, specified as:

Units	Height	Temperature	Speed of Sound	Air Pressure	Air Density
Metric (MKS)	Meters	Kelvin	Meters per second	Pascal	Kilograms per cubic meter
English (Velocity in ft/s)	Feet	Degrees Rankine	Feet per second	Pound-force per square inch	Slug per cubic foot
English (Velocity in kts)	Feet	Degrees Rankine	Knots	Pound-force per square inch	Slug per cubic foot

Programmatic Use
Block Parameter: units
Type: character vector
Values: 'Metric (MKS)' | 'English (Velocity in ft/s)' | 'English (Velocity in kts)'
Default: 'Metric (MKS)'

Specification — Atmosphere model type
1976 COESA-extended U.S. Standard Atmosphere (default) | MIL-HDBK-310 | MIL-STD-210C

Atmosphere model type, specified as 1976 COESA-extended U.S. Standard Atmosphere, MIL-HDBK-310, or MIL-STD-210C. For the MIL-HDBK-310 and MIL-STD-210C options:

MIL-HDBK-310	This selection is linked to the Non-Standard Day 310 block. See the block reference for more information. Selecting MIL-HDBK-310 enables the parameters **Atmospheric model type**, **Extreme parameter**, **Frequency of occurrence**, and **Altitude of extreme value**.
MIL-STD-210C	This selection is linked to the Non-Standard Day 210C block. See the block reference for more information. Selecting MIL-HDBK-310 enables the parameters **Atmospheric model type**, **Extreme parameter**, **Frequency of occurrence**, and **Altitude of extreme value**.

Dependencies

Selecting MIL-HDBK-310 or MIL-STD-210C enables these parameters:

- **Atmospheric model type**
- **Extreme parameter**
- **Frequency of occurrence**

- **Altitude of extreme value**

Programmatic Use
Block Parameter: `spec`
Type: character vector
Values: `'1976 COESA-extended U.S. Standard Atmosphere'` | `'MIL-HDBK-310'` | `'MIL-STD-210C'`
Default: `'1976 COESA-extended U.S. Standard Atmosphere'`

Atmospheric model type — Model type
`Profile` (default) | `Envelope`

Representation of atmospheric model type, specified as:

`Profile`	Realistic atmospheric profiles associated with extremes at specified altitudes. Recommended for simulation of vehicles vertically traversing the atmosphere or when the total influence of the atmosphere is needed.
`Envelope`	Uses extreme atmospheric values at each altitude. Recommended for vehicles only horizontally traversing the atmosphere without much change in altitude.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- Selecting `Profile` enables the **Attitude of extreme value** parameter.

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'Profile'` | `'Envelope'`
Default: `'Profile'`

Extreme parameter — Model type
`High temperature` (default) | `Low temperature` | `High density` | `Low density` | `High pressure` | `Low pressure`

Atmospheric parameter that is the extreme value.

Dependencies

- Selecting `MIL-HDBK-310` or `MIL-STD-210C` for the **Specification** parameter enables this parameter.
- The `High pressure` and `Low pressure` options appear only when **Atmospheric model type** is set to `Envelope`.

Programmatic Use
Block Parameter: `profile_var`
Type: character vector
Values: `'High temperature'` | `'Low temperature'` | `'High density'` | `'Low density'` | `'High pressure'` | `'Low pressure'`
Default: `'High temperature'`

Frequency of occurrence — Model type
`1%` (default) | `Extreme values` | `5%` | `10%` | `20%`

Percent of time the values would occur.

Dependencies

- Selecting MIL-HDBK-310 or MIL-STD-210C for the **Specification** parameter enables this parameter.
- Extreme values, 5%, and 20% are available only when Envelope is selected for **Atmospheric model type**.
- 1% and 10% are always available.

Programmatic Use
Block Parameter: profile_percent
Type: character vector
Values: 'Extreme values' | '1%' | '5%' | '10%' | '20%'
Default: '1%'

Altitude of extreme value — Geometric altitude
5 km (16404 ft) (default) | 10 km (32808 ft) | 20 km (65617 ft) | 30 km (98425 ft) | 40 km (131234 ft)

Geometric altitude at which the extreme values occur, specified as 5 km (16404 ft), 10 km (32808 ft), 20 km (65617 ft), 30 km (98425 ft), or 40 km (131234 ft).

Dependencies

This parameter appears if the **Atmospheric model type** is set to Profile.

Programmatic Use
Block Parameter: profile_alt
Type: character vector
Values: 5 km (16404 ft) | 10 km (32808 ft) | 20 km (65617 ft) | 30 km (98425 ft) | 40 km (131234 ft)
Default: 40 km (131234 ft)

Action for out-of-range input — Out-of-range block behavior

None (default) | Warning | Error

Out-of-range block behavior, specified as:

Action	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: action
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'Warning'

Version History
Introduced before R2006a

References

[1] *Global Climatic Data for Developing Military Products.* MIL-HDBK-310, Washington, D.C.: Department of Defense, 1987.

Extended Capabilities

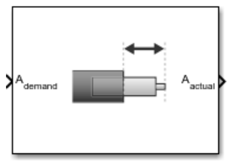
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

CIRA-86 Atmosphere Model | COESA Atmosphere Model | ISA Atmosphere Model | Non-Standard Day 210C

Nonlinear Second-Order Actuator

Implement second-order actuator with rate and deflection limits



Libraries:
Aerospace Blockset / Actuators

Description

The Second Order Nonlinear Actuator block outputs the actual actuator position using the input demanded actuator position and other dialog box parameters that define the system.

Ports

Input

A_{demand} — Demanded actuator position
scalar | array

Demanded actuator position, specified as a scalar or array.

Data Types: `double`

Output

A_{actual} — Actual actuator position
scalar | array

Actual actuator position, returned as a scalar or array.

Data Types: `double`

Parameters

Natural frequency — Natural frequency

1 (default) | scalar

Natural frequency of actuator, specified as a scalar double, in radians per second.

Programmatic Use
Block Parameter: `wn_fin`
Type: character vector
Values: scalar | double
Default: `'1'`

Damping ratio — Damping ratio

`0.3` (default) | scalar

Damping ratio of actuator, specified as a scalar double.

Programmatic Use
Block Parameter: `z_fin`
Type: character vector
Values: scalar | double
Default: `'0.3'`

Maximum deflection — Largest actuator position allowable

`20*pi/180` (default) | scalar

Largest actuator position allowable, specified as a scalar double, in the same units as demanded actuator position.

Programmatic Use
Block Parameter: `fin_max`
Type: character vector
Values: scalar | double
Default: `'20*pi/180'`

Minimum deflection — Smallest actuator position allowable

`-20*pi/180` (default) | scalar

Smallest actuator position allowable, specified as a scalar double, in the same units as demanded actuator position.

Programmatic Use
Block Parameter: `fin_min`
Type: character vector
Values: scalar | double
Default: `'-20*pi/180'`

Rate limit — Fastest speed allowable

`500*pi/180` (default) | scalar

Fastest speed allowable for actuator motion, specified as a scalar double, in the units of demanded actuator position per second.

Programmatic Use
Block Parameter: `fin_maxrate`
Type: character vector
Values: scalar | double
Default: `'500*pi/180'`

Initial position — Initial position

`0` (default) | scalar

Initial position of actuator, specified as a scalar double, in the same units as demanded actuator position.

- If the specified value is less than the value of **Minimum deflection**, the block sets the value of **Minimum deflection** as the initial position value.
- If the specified value is greater than the value of **Maximum deflection**, the block sets the value of **Maximum deflection** as the initial position value.

Programmatic Use
Block Parameter: `fin_act_0`
Type: character vector
Values: scalar | double
Default: `'0'`

Initial velocity — Initial velocity

`0` (default) | scalar

Initial velocity of actuator, specified as a scalar double, in the units of demanded actuator position per second.

If the absolute value of the specified value is greater than the absolute value of **Rate Limit**, this block sets the value of **Rate Limit** as the initial velocity value.

Programmatic Use
Block Parameter: `fin_act_vel`
Type: character vector
Values: scalar | double
Default: `'0'`

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

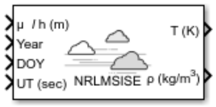
Linear Second-Order Actuator

Topics

"Explore the NASA HL-20 Model" on page 1-5

NRLMSISE-00 Atmosphere Model

Implement mathematical representation of 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The NRLMSISE-00 Atmosphere Model block implements the mathematical representation of the 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere (NRLMSISE-00) of the MSIS® class model. This block calculates the neutral atmosphere empirical model from the surface to lower exosphere (0 to 1,000,000 meters). When configuring the block for this calculation, you can also take into account the anomalous oxygen, which can affect the satellite drag above 500,000 meters.

Limitations

- This block has the limitations of the NRLMSISE-00 model. For more information, see https://ccmc.gsfc.nasa.gov/.
- This block is valid only for altitudes between 0 and 1,000,000 meters (1,000 kilometers).
- The F107 and F107A values used to generate the model correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun rather than the radio flux at 1 AU. This site provide both classes of values: https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

Ports

Input

Port_1 — Geodetic latitudes, longitude, and altitude
three-element matrix

Geodetic latitudes, in degrees, longitude, in degrees, and altitude, in selected length units, specified as three-element matrix.

Data Types: `double`

Port_2 — Years
array

N years, specified as an array.

Data Types: `double`

Port_3 — Days
array

N days of a year (1 to 365 (or 366)), specified as an array.

Data Types: `double`

Port_4 — Seconds
array

N seconds in a day, specified as an array, in universal time (UT).

Data Types: `double`

Port_5 — Local apparent solar time
array

N local apparent solar timed, specified as an array, in hours.

Data Types: `double`

Port_6 — 81-day average F10.7 flux
`150` (default) | array

N 81-day averages of F10.7 flux, centered on day of year (doy), specified as an array.

Data Types: `double`

Port_7 — Daily average F10.7 flux
`150` (default) | array

N daily F10.7 fluxes for previous days, specified as an array.

Data Types: `double`

Port_8 — Magnetic index information
N-by-7 array

Magnetic index information, specified as an *N*-by-7. If you specify *magneticIndex*, you must also specify *f107Average* and *f107Daily*. The magnetic index information consists of:
Daily magnetic index (AP)
3 hour AP for current time
3 hour AP for 3 hours before current time
3 hour AP for 6 hours before current time
3 hour AP for 9 hours before current time
Average of eight 3 hour AP indices from 12 to 33 hours before current time
Average of eight 3 hour AP indices from 36 to 57 hours before current time

The effects of daily magnetic index are not large or established below 80,000 m. As a result, the block sets the default value to `4`. It sets all other indices to `0` by default. See the limitations in "Limitations" on page 5-615 for more information.

Data Types: `double`

Port_9 — Flags
array of 23

Flags, specified as an array of 21, to enable or disable particular variations for the outputs.

Field	Description
`Flags(1)`	F10.7 effect on mean
`Flags(2)`	Independent of time
`Flags(3)`	Symmetrical annual
`Flags(4)`	Symmetrical semiannual
`Flags(5)`	Asymmetrical annual
`Flags(6)`	Asymmetrical semiannual
`Flags(7)`	Diurnal
`Flags(8)`	Semidiurnal
`Flags(9)`	Daily AP. If you set this field to -1, the block uses the entire matrix of magnetic index information (APH) instead of `APH(:,1)`
`Flags(10)`	All UT, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

Data Types: `double`

Output

Port_1 — Temperature
N-by-2 array

Temperature values, returned in a *N*-by-2 array of values, in selected temperature units. The first column contains exospheric temperatures, the second column contains temperature at altitude.

Data Types: `double`

Port_2 — Densities
N-by-9 array

Density values, returned in a *N*-by-9 array, in selected density units.

Density	Description
`Density(1)`	Density of He
`Density(2)`	Density of O
`Density(3)`	Density of N2
`Density(4)`	Density of O2
`Density(5)`	Density of Ar
`Density(6)`	Total mass density `Density(6)`, total mass density, is defined as the sum of the mass densities of He, O, N2, O2, Ar, H, and N. Optionally, `Density(6)` can include the mass density of anomalous oxygen making `Density(6)`, the effective total mass density for drag.
`Density(7)`	Density of H
`Density(8)`	Density of N
`Density(9)`	Anomalous oxygen number density

Data Types: `double`

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English`

Input and output units, specified as:

Units	Temperature	Height	Density
`Metric (MKS)`	Kelvin	Meters	kg/m^3, some density outputs $1/m^3$
`English`	Rankine	Feet	lbm/ft^3, some density outputs $1/ft^3$

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Input local apparent solar time — Apparent solar time

`off` (default) | `on`

Select this check box to input the local apparent solar time, in hours. Otherwise, the block inputs the default value.

Programmatic Use
Block Parameter: `lst input`
Type: character vector

Values: `'off'` | `'on'`
Default: `'off'`

Input flux and magnetic index information — Daily F10.7 flux for previous day and magnetic index information

`off` (default) | `on`

Select this check box to input the 81-day average of F10.7, the daily F10.7 flux for the previous day, and the array of 7 magnetic index information (see the `aph` argument in the `atmosnrlmsise00` function). Otherwise, the block inputs the default value of `150` for F10.7 and F10.7a, and `[4, 0, 0, 0, 0, 0, 0]` for geomagnetic index.

Programmatic Use
Block Parameter: `flux_ap_input`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Source for flags — Variation flag source

`Internal` (default) | `External`

Variation flag source, specified as `Internal` or `External`. If you specify `External`, specify the variation flag as an array of 23. If you specify `Internal`, the flag source is internal to the block.

Dependencies

Setting **Source for flags** to `Internal` enables the **Flags** parameter.

Programmatic Use
Block Parameter: `flags_input`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Flags — Variation flags

`ones(1,23)` (default)

Variation flag, specified as an array of 23 (`ones(1,23)`). This parameter applies only when **Source for flags** has a value of `Internal`. You can specify one of the following values for a field. The default value for each field is `1`.

- 0.0

  Removes that value's effect on the output.
- 1.0

  Applies the main and the cross-term effects of that value on the output.
- 2.0

  Applies only the cross-term effect of that value on the output.

The array has the following fields.

Field	Description
`Flags(1)`	F10.7 effect on mean
`Flags(2)`	Independent of time
`Flags(3)`	Symmetrical annual
`Flags(4)`	Symmetrical semiannual
`Flags(5)`	Asymmetrical annual
`Flags(6)`	Asymmetrical semiannual
`Flags(7)`	Diurnal
`Flags(8)`	Semidiurnal
`Flags(9)`	Daily AP. If you set this field to -1, the block uses the entire matrix of magnetic index information (APH) instead of `APH(:,1)`
`Flags(10)`	All UT, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

Dependencies

Setting **Source for flags** to `Internal` enables the **Flags** parameter.

Programmatic Use

Block Parameter: `flags`
Type: character vector
Values: `'ones(1,23)'`
Default: `'ones(1,23)'`

Include anomalous oxygen number density in total mass density — Anomalous oxygen

`off` (default) | `on`

Select this check box to take into account the anomalous oxygen when calculating the neutral atmosphere empirical model from the surface to lower exosphere (0 to 1,000,000 meters). Taking into account this number can affect the satellite drag above 500,000 meters.

Programmatic Use
Block Parameter: `oxygen_in`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

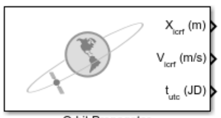
CIRA-86 Atmosphere Model | COESA Atmosphere Model | Solar Flux and Geomagnetic Index | ISA Atmosphere Model

External Websites

https://ccmc.gsfc.nasa.gov/
https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

Orbit Propagator

Propagate orbit of one or more spacecraft



Orbit Propagator
Kepler (unperturbed)

Libraries:
Aerospace Blockset / Spacecraft / Spacecraft Dynamics

Description

The Orbit Propagator block propagates the orbit of one or more spacecraft by a propagation method. The library contains two versions of the Orbit Propagator block preconfigured for these propagation methods:

Solar flux pressure, specified as a scalar.

- Kepler (unperturbed) — Kepler universal variable formulation (quicker)
- Numerical (high precision) — More accurate

The size of the provided initial conditions determines the number of spacecraft being modeled. If you supply more than one value for a parameter in the **Orbit** tab, the block outputs a constellation of satellites. Any parameter with a single provided value is expanded and applied to all the satellites in the constellation. For example, if you provide a single value for all the parameters on the block except **True anomaly**, which contains six values, the block creates a constellation of six satellites, varying true anomaly only.

The block applies the same expansion behavior to input port **A_icrf** (applied acceleration). This port accepts either a single value expanded to all spacecraft being modeled, or individual values to apply to each spacecraft.

For more information on the propagation methods the Orbit Propagator block uses, see “Orbit Propagation Methods” on page 5-671.

You can define initial orbital states in the **Orbit** tab as:

- A set of orbital elements
- Position and velocity state vectors in International Celestial Reference Frame (ICRF) or fixed-frame coordinate systems.

The block uses quaternions, which are defined using the scalar-first convention.

For more information on the coordinate systems the Orbit Propagator block uses, see “Coordinate Systems” on page 5-669.

Atmospheric Drag

To help model the drag on spacecraft for high precision orbit propagation, the Orbit Propagator block supports drag. Atmospheric drag affects spacecraft flying at low Earth orbit (LEO); it is less relevant further away from Earth. For the atmospheric drag equation, see “Atmospheric Drag” on page 5-1208.

Third Body Gravity Effects

To include the effects of point-mass “Third Body” on page 5-657 gravity, select which third bodies to include. These calculations require an **Ephemeris model** and data. For the third body contribution equation, see “Third Body” on page 5-1208.

Solar Radiation Pressure

To include the effects of solar radiation pressure (“SRP” on page 5-664) on the spacecraft, provide an eclipse fraction or specify a shadow model to compute the fraction. Consider taking solar radiation pressure into account when atmosphere drag is negligible and pointing accuracy requirements are strict. For the solar radiation pressure equation, see “Solar Radiation Pressure” on page 5-1209.

Ports

Input

$\mathbf{A_{icrf}}$ — Applied acceleration
3-element vector | *m*-3 array

Acceleration applied to the spacecraft with respect to the port coordinate system (ICRF or fixed-frame), specified as a 3-element vector or *m*-by-3 array, at the current time step.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Input external accelerations** check box.

Data Types: `double`

φθψ — Moon libration angles
3-element vector

Moon libration angles for transformation between the ICRF and Moon-centric fixed-frame using the Moon-centric Principal Axis (PA) system, specified as a 3-element vector. To get these values, use the Moon Libration block.

Note The fixed-frame used by this block when **Central body** is set to `Moon` is the Mean Earth/pole axis (ME) system. For more information, see “Algorithms” on page 5-669.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Set **Central body** to `Moon`.
- Select the **Input Moon libration angles** check box.

Data Types: `double`

αδW — Right ascension, declination, and rotation angle
3-element vector

Central body spin axis instantaneous right ascension, declination, and rotation angle, specified as a 3-element vector. This port is available only for custom central bodies.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Port`.

Data Types: `double`

m — Spacecraft mass used by atmospheric drag calculation
scalar | vector of size *numSat*

Spacecraft mass used by atmospheric drag calculation, specified as scalar or vector of size *numSat*. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Mass source** parameter to `Dialog`.

Data Types: `double`

ρ — Atmospheric density
scalar

Atmospheric density to calculate acceleration due to atmospheric drag.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Port`.

Data Types: `double`

F107a — 81-day average Ottawa F10.7 cm solar flux
scalar

81-day average Ottawa F10.7 cm solar flux, centered on the current day specified in **Start date/time**. These F107 Average values correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. This site provides both classes of values:

https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Data Types: `double`

F107 — Daily Ottawa F10.7 cm solar flux
scalar

Daily Ottawa F10.7 cm solar flux, centered on the current day specified in **Start date/time**. The f107Daily values do not correspond to the radio flux at 1 AU. This site provides both classes of values:

https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Data Types: `double`

aph — Daily magnetic index information
N-by-7 array

Daily magnetic index information (aph), specified as an *N*-by-7 array. The magnetic index information consists of:
Daily magnetic index (AP)
3 hour AP for current time
3 hour AP for 3 hours before current time
3 hour AP for 6 hours before current time
3 hour AP for 9 hours before current time
Average of eight 3 hour AP indices from 12 to 33 hours before current time
Average of eight 3 hour AP indices from 36 to 57 hours before current time

The effects of daily magnetic index are not large or established below 80,000 m. For more information, see Limitations on NRLMSISE-00 Atmosphere Model.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Data Types: `double`

Flags — Variation flags
array of 23

Variation flags, specified as an array of 23, to enable or disable particular variations for the outputs. You can specify one of the following values for a field. The default value for each field is `1`.

- 0.0

  Removes the value effect on the output.
- 1.0

  Applies the main and the cross-term effects of that value on the output.
- 2.0

  Applies only the cross-term effect of that value on the output.

Field	Description
`Flags(1)`	F10.7 effect on mean
`Flags(2)`	Independent of time
`Flags(3)`	Symmetrical annual
`Flags(4)`	Symmetrical semiannual
`Flags(5)`	Asymmetrical annual
`Flags(6)`	Asymmetrical semiannual
`Flags(7)`	Diurnal
`Flags(8)`	Semidiurnal
`Flags(9)`	Daily AP. If you set this field to -1, the block uses the entire matrix of magnetic index information (APH) instead of `APH(:,1)`
`Flags(10)`	All UT, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations

Field	Description
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Flags source** parameter to `Port`.

Data Types: `double`

C_d — Atmospheric drag coefficient
scalar | vector of size *numSat*

Atmospheric drag coefficient, specified as a scalar or vector of size *numSat*.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag coefficient source** parameter to `Port`.

Data Types: `double`

A_d — Atmospheric drag area
scalar | vector of size *numSat*

Atmospheric drag area, specified as a scalar or vector of size *numSat*.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag area source** parameter to `Port`.

Data Types: `double`

$R_{cb, 3}$ — Position of one or more custom bodies
3-element vector | *numCustom3rdBodies*x3-element vector

Position of one or more custom bodies with respect to the current central body, in the ICRF coordinate frame. *numCustom3rdBodies* is the number of custom bodies. For more information, see **Custom gravitational parameter (m^3/s^2)**.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Include third body point-mass gravity** check box.
- Select the **Custom** check box.

Data Types: `double`

Fraction — Fraction of solar disk
between `0` (full eclipse) and `1` (full sunlight) | scalar | *numSat*

Fraction of solar disk visible from the spacecraft position. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.
- Set **Eclipse fraction source** to `Port`.

Data Types: `double`

R_C — Reflectivity coefficient
scalar or *numSat* of values in the range $1 \le R_C \le 2$

Reflectivity coefficient, specified as a scalar or *numSat* of values in the range $1 \le R_C \le 2$. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.
- Set the **Reflectivity coefficient source** parameter to `Port`.

Data Types: `double`

A_{srp} — Cross section area of the spacecraft
scalar | *numSat*

Cross section area of the spacecraft seen by the Sun, specified as a scalar or *numSat*. The block uses this value to calculate the acceleration due to solar radiation pressure.

Dependencies

To enable this port:

- Set **Propagation method** parameter to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.
- Set **SRP area source** parameter to `Port`.

Data Types: `double`

Output

X_{icrf} — Position of spacecraft
3-element vector | *numSat*-by-3

Position of the spacecraft with respect to (ICRF or fixed-frame), returned as a 3-element vector or *numSat*-by-3 array, where *m* is number of spacecraft, at the current time step. The size of the initial conditions provided in the **Orbit** tab control the port dimension. *numSat* is the number of spacecraft.

Data Types: `double`

V_{icrf} — Velocity
3-element vector | *numSat*-by-3 array

Velocity of the spacecraft with respect to ICRF or fixed-frame, returned as a 3-element vector or *numSat*-by-3 array, at the current time step. *numSat*-by-3 array. The size of the initial conditions provided in the **Orbit** tab control the port dimension.

Data Types: `double`

$q_{icrf2ff}$ — Transformation
4-element quaternion (scalar first)

Transformation between the ICRF coordinate system and fixed-frame, returned as a 4-element vector (scalar first), at the current time step.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Output quaternion (ICRF to Fixed-frame)** check box.

Data Types: `double`

t_{utc} — Time at current time step
scalar | 6-element vector

Time at current time step, returned as a:

- scalar — If you specify the **Start data/time** parameter as a Julian date.
- 6-element vector — If you specify the **Start data/time** parameter as a Gregorian date with six elements (year, month, day, hours, minutes, seconds).

This value is equal to the **Start date/time** parameter value + *the elapsed simulation time*.

Dependencies

To enable this parameter, select the **Output current date/time (UTC Julian date)** check box.

Data Types: `double`

Parameters

Main

Propagation method — Orbit propagation method

`Kepler (unperturbed)` | `Numerical (high precision)`

Orbit propagation method, specified as:

- `Kepler (unperturbed)` — Uses a universal variable formulation of the Kepler problem to determine the spacecraft position and velocity at each time step. This method is faster than `Numerical (high precision)`.
- `Numerical (high precision)` — Determine the spacecraft position and velocity at each time step using numerical integration. This option models central body gravity based on the settings in the **Central body** tab. This method is more accurate than `Kepler (unperturbed)`, but slower.

Programmatic Use
Block Parameter: `propagator`
Type: character vector
Values: `'Kepler (unperturbed)'` | `'Numerical (high precision)'`
Default: `'Kepler (unperturbed)'`

Input external accelerations — Input additional accelerations

off (default) | on

To enable additional external accelerations to be included in the integration of the spacecraft equations of motion, select this check box. Otherwise, clear this check box.

Dependencies

To enable this check box, set **Propagation method** to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `accelIn`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

External acceleration coordinate frame — Frame for additional accelerations

`ICRF` (default) | `Fixed-frame`

Input additional accelerations, specified as `ICRF` or `Fixed-frame`. These accelerations are included in integration of the spacecraft equations of motion.

Dependencies

To enable this parameter:

- Set **Propagation method** to `Numerical (high precision)`
- Select the **Input external accelerations** check box

Programmatic Use

Block Parameter: `accelFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'`
Default: `'ICRF'`

State vector output coordinate frame — Port coordinate frame

`ICRF` (default) | `Fixed-frame`

Coordinate frame for output ports, specified as `ICRF` or `Fixed-frame`. These port labels are affected:

- Output port **X**
- Output port **V**

Dependencies

To enable this parameter, set **Propagation method** to `Numerical (high precision)`.

Programmatic Use

Block Parameter: `outportFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'`
Default: `'ICRF'`

Start date/time (UTC Julian date) — Initial start time for simulation

`juliandate (2020, 1, 1, 12, 0, 0)` (default) | valid scalar Julian date | valid Gregorian date including year, month, day, hours, minutes, seconds as 6-element vector

Initial start date and time of simulation, specified as a Julian or Gregorian date. The block defines initial conditions using this value.

Tip To calculate the Julian date, use the `juliandate` function.

Programmatic Use

Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1, 12, 0, 0)'` | valid scalar Julian date | valid Gregorian date including year, month, day, hours, minutes, seconds as 6-element vector
Default: `'juliandate(2020, 1, 1, 12, 0, 0)'`

Output current date/time (UTC Julian date) — Add output port t_{utc}

on (default) | off

To output the current date or time, select this check box. Otherwise, clear this check box.

Programmatic Use

Block Parameter: `dateOut`
Type: character vector
Values: `'off'` | `'on'`

Default: `'off'`

Mass source — Mass source

`Dialog` (default) | `Port`

Mass source, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `massSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Mass — Spacecraft mass used by atmospheric drag calculation

`4.0` (default) | scalar | vector of size *numSat*

Spacecraft mass used by atmospheric drag calculation, specified as scalar or vector of size *numSat*. *numSat* is the number of spacecraft.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Mass source** to `Dialog`.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'4.0'`

Action for out-of-range input — Out-of-range block behavior

`Warning` (default) | `Error` | `None`

Out-of-range block behavior, specified as follows:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.

Action	Description
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Orbit

Define the initial states of the space craft.

Initial state format — Input method for initial states of orbit

`Orbital elements` (default) | `ICRF state vector` | `Fixed-frame state vector`

Input method for initial states of orbit, specified as `Orbital elements`, `ICRF state vector`, or `Fixed-frame state vector`.

Dependencies

Available options are based on **Propagation method** settings:

Kepler (unperturbed)	Numerical (high precision)
Orbital elements	Orbital elements
ICRF state vector	ICRF state vector
—	Fixed-frame state vector

Programmatic Use
Block Parameter `stateFormatKep` when `propagator` is set to `Kepler (unperturbed)`, `stateFormatNum` when `propagator` is set to `Numerical (high precision)`
Type: character vector
Values: `'Orbital elements'` | `'ICRF state vector'` when `propagator` is set to `'Kepler (unperturbed)'` | `'Orbital elements'` | `'ICRF state vector'` | `'Fixed-frame state'` when `propagator` is set to `'Numerical (high precision)'`
Default: `'Orbital elements'`

Orbit Type — Orbit classification

`Keplerian` (default) | `Elliptical equatorial` | `Circular` | `Circular equatorial`

Orbit classification, specified as:

- `Keplerian` — Model elliptical orbits using six standard Keplerian orbital elements.
- `Elliptical equatorial` — Define an equatorial orbit, where inclination is 0 or 180 degrees and the right ascension of the ascending node is undefined.
- `Circular` — Define a circular orbit, where eccentricity is 0 and the argument of periapsis is undefined.
- `Circular equatorial` — Define a circular orbit, where eccentricity is 0 or 10 degrees. Argument of periapsis and the right ascension of the ascending node are undefined.

Dependencies

To enable this parameter, set **Initial state format** to `Orbital elements`.

Programmatic Use

Block Parameter: `orbitType`
Type: character vector
Values: `'Keplerian'` | `'Elliptical equatorial'` | `'Circular inclined'` | `'Circular equatorial'`
Default: `'Keplerian'`

Semi-major axis — Half of major axis of ellipse

6786000 (default) | 1D array of size *numSat*

Half of ellipsis major axis, specified as a 1D array whose size is the number of spacecraft.

- For parabolic orbits, this block interprets this parameter as the periapsis radius (distance from periapsis to the focus point of orbit).
- For hyperbolic orbits, this block interprets this parameter as the distance from periapsis to the hyperbola center.

Dependencies

To enable this parameter, set **Initial state format** to `Orbital elements`.

Programmatic Use

Block Parameter: `semiMajorAxis`
Type: character vector
Values: scalar | 1D array of size *m*, number of spacecraft
Default: `'6786000'`

Eccentricity — Deviation of orbit

0.01 (default) | scalar | value between `0` and `1`, or greater than `1` for Keplerian orbit type | 1D array of size *numSat*

Deviation of the orbit from a perfect circle, specified as a scalar or 1D array of size that is number of spacecraft.

If **Orbit** type is set to `Keplerian`, value can be:

- `0` for circular orbit
- Between `0` and `1` for elliptical orbit
- `1` for parabolic orbit
- Greater than `1` for hyperbolic orbit

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`.
- **Orbit type** to `Keplerian` or `Elliptical equatorial`.

Programmatic Use
Block Parameter: `eccentricity`
Type: character vector
Values: `0.01` | scalar | value between `0` and `1`, or greater than `1` for Keplerian orbit type | 1D array of size *numSat*
Default: `'0.01'`

Inclination (deg) — Tilt angle of orbital plane

50 (default) | scalar | 1D array of size *numSat* | degrees between 0 and 180 | radians between 0 and pi

Vertical tilt of the ellipse with respect to the reference plane measured at the ascending node, specified as a scalar or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft.

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`
- **Orbit type** to `Keplerian` or `Circular inclined`

Programmatic Use
Block Parameter: `inclination`
Type: character vector
Values: 50 | scalar | 1D array of size *numSat* | degrees between 0 and 180 | radians between 0 and pi
Default: `'50'`

RAAN (deg) — Angular distance in equatorial plane

95 (default) | scalar value between `0` and `360` | 1D array of size *numSat*

Right ascension of ascending node (RAAN), specified as a scalar value between `0` and `360` or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft. RAAN is the angular distance along the reference plane from the ICRF *x*-axis to the location of the ascending node (the point at which the spacecraft crosses the reference plane from south to north).

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`.
- **Orbit type** to `Keplerian` or `Circular inclined`.

Programmatic Use
Block Parameter: `raan`
Type: character vector
Values: `95` | scalar value between `0` and `360` | 1D array of size *m* number of spacecraft
Default: `'95'`

Argument of periapsis (deg) — Angle from spacecraft ascending node to periapsis

93 (default) | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *m*, number of spacecraft

Angle from the spacecraft ascending node to periapsis (closest point of orbit to the central body), specified as a 1D array of size *m* that is number of spacecraft, in specified units.

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`
- **Orbit type** to `Keplerian`

Programmatic Use

Block Parameter: `argPeriapsis`
Type: character vector
Values: `'95'` | scalar value between `0` and `360` | 1D array of size *numSat*
Default: `'93'`

True anomaly — Angle between periapsis and initial position of spacecraft

203 (default) | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*

Angle between periapsis (closest point of orbit to the central body) and the initial position of spacecraft along its orbit at **Start date/time**, specified as a scalar or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft.

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`.
- **Orbit type** to `Keplerian` or `Elliptical inclined`.

Programmatic Use

Block Parameter: `trueAnomaly`
Type: character vector
Values: `'203'` | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*
Default: `'203'`

Argument of latitude (deg) — Angle between ascending node and initial position of spacecraft

200 (default) | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*

Angle between the ascending node and the initial position of spacecraft along its orbit at **Start date/time**, specified as a scalar or 3-element vector or 1D array of size number of spacecraft, in specified units.

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`.
- **Orbit Type** to `Circular inclined`.

Programmatic Use
Block Parameter: `argLat`
Type: character vector
Values: `'200'` | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*
Default: `'200'`

Longitude of periapsis (deg) — Angle between ICRF *x*-axis and eccentricity vector

100 (default) | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*

Angle between the ICRF x-axis and the eccentricity vector, specified as a scalar or 3-element vector or 1D array of size number of spacecraft, in specified units.

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`.
- **Orbit type** to `Elliptical equatorial`.

Programmatic Use
Block Parameter: `lonPeriapsis`
Type: character vector
Values: 100 | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *m*, number of spacecraft
Default: `'100'`

True longitude (deg) — Angle between ICRF *x*-axis and initial position of spacecraft

150 (default) | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*

Angle between the ICRF x-axis and the initial position of spacecraft along its orbit at **Start date/time**, specified as a scalar or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft.

Dependencies

To enable this parameter, set:

- **Initial state format** to `Orbital elements`.
- **Orbit type** to `Circular equatorial`.

Programmatic Use
Block Parameter: `trueLon`
Type: character vector
Values: `'150'` | scalar | degrees between `0` and `360` | radians between `0` and `2*pi` | 1D array of size *numSat*
Default: `'150'`

ICRF position — Cartesian position vector of spacecraft

`[3649700.0 3308200.0 -4676600.0]` (default) | 3-element vector | | *numSat*-by-3 array

Cartesian position vector of spacecraft in ICRF coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or *numSat*-by-3 array for multiple spacecraft. *numSat* is the number of spacecraft.

Dependencies

To enable this parameter, set **Initial state format** to `ICRF state vector`.

Programmatic Use

Block Parameter: `inertialPosition`
Type: character vector
Values: `[3649700.0 3308200.0 -4676600.0]` | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft
Default: `'[3649700.0 3308200.0 -4676600.0]'`

ICRF velocity — Cartesian velocity vector of spacecraft

`[-2750.8 6666.4 2573.4]` (default) | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft

Cartesian velocity vector of spacecraft in ICRF coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft.

Dependencies

To enable this parameter, set **Initial state format** to `ICRF state vector`.

Programmatic Use

Block Parameter: `inertialVelocity`
Type: character vector
Values: `[-2750.8 6666.4 2573.4]` | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft
Default: `'[-2750.8 6666.4 2573.4]'`

Fixed-frame position — Position vector of spacecraft

[-4142689.0 -2676864.7 -4669861.6] (default) | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft

Cartesian position vector of spacecraft in fixed-frame coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- set **Initial state format** to `Fixed-frame state vector`.

Programmatic Use

Block Parameter: `fixedPosition`
Type: character vector
Values: `'[-4142689.0 -2676864.7 -4669861.6]'` | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft
Default: `'[-2750.8 6666.4 2573.4]'`

Fixed-frame velocity — Velocity vector of spacecraft

[1452.7 -6720.7 2568.1] (default) | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft

Cartesian velocity vector of spacecraft in fixed-frame coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Initial state format** to `Fixed-frame state vector`.

Programmatic Use

Block Parameter: `fixedVelocity`
Type: character vector
Values: `'[1452.7 -6720.7 2568.1]'` | 3-element vector for single spacecraft or 2-D array of size *m*-by-3 array of multiple spacecraft
Default: `'[1452.7 -6720.7 2568.1]'`

Central Body

Configure the central body environment around which the spacecraft orbits.

Central body — Celestial body around which spacecraft orbits

`Earth` (default) | `Moon` | `Mercury` | `Venus` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Sun` | `Custom`

Celestial body, specified as `Earth`, `Moon`, `Mercury`, `Venus`, `Mars`, `Jupiter`, `Saturn`, `Uranus`, `Neptune`, `Sun`, or `Custom`, around which the spacecraft defined in the **Orbit** tab orbits.

Programmatic Use

Block Parameter: `centralBody`
Type: character vector
Values: `'Earth'` | `'Moon'` |`'Mercury'` | `'Venus'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Sun'` | `'Custom'` |
Default: `'Earth'`

Gravitational potential model — Control gravity model for central body

`Spherical harmonics` when **Central body** set to `Earth`, `Moon`, `Mars`, or `Custom`, Oblate ellipsoid when **Central body** set to `Mercury`, `Venus`, `Jupiter`, `Saturn`, `Uranus`, or `Neptune` (default) | `None` | `Point-mass` | `Oblate ellipsoid (J2)`

Control the gravity model for the central body, specified as `Spherical harmonics`, `Point-mass`, or `Oblate ellipsoid (J2)`.

Dependencies

To enable this parameter, set **Propagation method** to `Numerical (high precision)`. Available options are based on **Central body** settings:

Earth, Moon, Mars, or Custom	Mercury, Venus, Jupiter, Saturn, Uranus, or Neptune
`None`	`None`
`Spherical harmonics`	`Oblate ellipsoid (J2)`
`Point-mass`	`Point-mass`
`Oblate ellipsoid (J2)`	—

Programmatic Use
Block Parameter: `gravityModel` when `centralBody` set to `'Earth'`, `'Moon'`, `'Mars'`, or `'Custom'` | `gravityModelnoSH` when `centralBody` set to `Mercury`, `Venus`, `Jupiter`, `Saturn`, `Uranus`, or `Neptune`
Type: character vector
Values: `'Spherical harmonics'` | `'None'` | `'Point-mass'` | `'Oblate ellipsoid (J2)'` when `centralBody` set to `'Earth'`, `'Moon'`, `'Mars'`, or `'Custom'`; `'Point-mass'` | `'Oblate ellipsoid (J2)'` when `centralBody` set to `Mercury`, `Venus`, `Jupiter`, `Saturn`, `Uranus`, or `Neptune`
Default: `'Spherical harmonics'` when `centralBody` set to `'Earth'`, `'Moon'`, `'Mars'`, or `'Custom'`; `'Oblate ellipsoid (J2)'` when `centralBody` set to `Mercury`, `Venus`, `Jupiter`, `Saturn`, `Uranus`, or `Neptune`

Spherical harmonic model — Spherical harmonic model

`EGM2008` for **Central body** set to `Earth`, `LP-100K` for **Central body** set to `Moon`, `GMM2B` for **Central body** set to `Mars`, (default) | `EGM96` | `EIGEN-GL04C` | `LP-165P`

Spherical harmonic gravitational potential model, specified according to the specified **Central body**.

Dependencies

To enable this parameter, set **Propagation method** to `Numerical (high precision)`. Available options are based on **Central body** settings:

Central body	Spherical Harmonic Model Option
Earth	EGM2008, EGM96, or EIGEN-GL04C
Moon	LP-100K or LP-165P
Mars	GMM2B

Programmatic Use
Block Parameter: `'earthSH'` when `centralBody` set to `'Earth'` | `'moonSH'` when `centralBody` set to `'Moon'` | `'marsSH'` when `centralBody` set to `'Mars'`
Type: character vector
Values: `'EGM2008'` | `'EGM96'` | `'EIGEN-GL04C'` when `centralBody` set to `'earthSH'`; `'LP-100K'` | `'LP-165P'` when `centralBody` set to `'moonSH'`; `'GMM2B'` when `centralBody` set to `'marsSH'`
Default: `'Spherical harmonics'`

Spherical harmonic coefficient file — Harmonic coefficient MAT-file

`aerogmm2b.mat` (default) | harmonic coefficient MAT-file

Harmonic coefficient MAT-file that contains definitions for a custom planetary model, specified as a character vector or string.

This file must contain:

Variable	Description
Re	Scalar of planet equatorial radius in meters (m).
GM	Scalar of planetary gravitational parameter in meters cubed per second squared (m^3/s^2).
degree	Scalar of maximum degree.
C	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *C*.
S	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *S*.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Gravitational potential model** to `Spherical harmonics`.

Programmatic Use

Block Parameter: `shFile`
Type: character vector
Values: `'aerogmm2b.mat'` | harmonic coefficient MAT-file
Default: `'aerogmm2b.mat'`

Degree — Degree of harmonic model

`120` (default) | scalar | maximum of 2159

Degree of harmonic model, specified as a double scalar:

Planet Model	Recommended Degree	Maximum Degree
`EGM2008`	120	2159
`EGM96`	70	360
`LP100K`	60	100
`LP165P`	60	165
`GMM2B`	60	80
`EIGENGL04C`	70	360
`Custom`	—	Maximum degree is extracted from the spherical harmonic coefficient file.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.

- **Central body** to `Earth`, `Moon`, `Mars`, or `Custom`.
- **Gravitational potential model** to `Spherical harmonics`.

Programmatic Use
Block Parameter: `shDegree`
Type: character vector
Values: `'80'` | scalar
Default: `'80'`

Use Earth orientation parameters (EOPs) — Use Earth orientation parameters

on (default) | off

Select this check box to use Earth orientation parameters for the transformation between the ICRF and fixed-frame coordinate systems. Otherwise, clear this check box.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Earth`.

Additionally, it must satisfy one of these criteria:

- **Gravitational potential model** is set to either `Spherical harmonics` or `Oblate ellipsoid (J2)`.
- **External acceleration coordinate frame** is set to `Fixed-frame`.
- **State vector output coordinate frame** is set to `Fixed-frame`.

Programmatic Use
Block Parameter: `useEOPs`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

IERS EOP data file — Earth orientation data

`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

Dependencies

To enable this parameter:

- Select the **Use Earth orientation parameters (EOPs)** to check box.
- Set **Propagation method** to `Numerical (high precision)`.
- Set **Central body** to `Earth`.

Programmatic Use
Block Parameter: `eopFile`
Type: character vector
Values: `'aeroiersdata.mat'` | `MAT-file`

Default: `'aeroiersdata.mat'`

Input Moon libration angles — Moon libration angle rate

off (default) | on

To specify libration angles (φ θ ψ) for Moon orientation, select this check box.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Moon`.

Programmatic Use
Block Parameter: `useMoonLib`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Output quaternion (ICRF to Fixed-frame) — Add output transformation quaternion port

off (default) | on

To add output transformation quaternion port for the quaternion transformation from the ICRF to the Fixed-frame coordinate system, select this check box.

Dependencies

To enable this check box, set **Propagation method** to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `outputTransform`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Central body spin axis source — Central body spin source

`Port` (default) | `Dialog`

Central body spin axis, specified as `Port` or `Dialog`. The block uses the spin axis to calculate the transformation from the ICRF to the fixed-frame coordinate system for the custom central body.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.

Programmatic Use
Block Parameter: `cbPoleSrc`
Type: character vector
Values: `'Port'` | `'Dialog'`

Default: `'Port'`

Spin axis right ascension (RA) at J2000 (deg) — Right ascension of central body spin axis at J2000

`317.68143` (default) | double scalar

Right ascension of central body spin axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRA`
Type: character vector
Values: `'317.68143'` | double scalar
Default: `'317.68143'`

Spin axis RA rate (deg/century) — Right ascension rate of central body spin axis

`-0.1061` (default) | double scalar

Right ascension rate of the central body spin axis, specified as a double scalar, in specified angle units/century.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRARate`
Type: character vector
Values: `'-0.1061'` | double scalar
Default: `'-0.1061'`

Spin axis declination (Dec) at J2000 (deg) — Declination of central body spin axis at J2000

`52.88650` (default) | double scalar

Declination of the central body spin axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbDec`
Type: character vector
Values: `'52.88650'` | double scalar
Default: `'52.88650'`

Spin axis Dec rate (deg/century) — Declination rate of central body spin axis

`-0.0609` (default) | double scalar

Declination rate of the central body spin axis, specified as a double scalar, in specified angle units/century.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbDecRate`
Type: character vector
Values: `'-0.0609'` | double scalar
Default: `'-0.0609'`

Initial rotation angle at J2000 (deg) — Rotation angle of central body *x*-axis

`176.630` (default) | double scalar

Rotation angle of the central body x axis with respect to the ICRF *x*-axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar, in specified angle units.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRotAngle`
Type: character vector
Values: `'176.630'` | double scalar
Default: `'176.630'`

Rotation rate (deg/day) — Rotation rate of central body *x*-axis

`350.89198226` (default) | double scalar

Rotation rate of the central body x axis with respect to the ICRF *x*-axis (2451545.0 JD, 2000 Jan 1 12:00:00 UTC), specified as a double scalar, specified angle units/day.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRotRate`
Type: character vector
Values: `'350.89198226'` | double scalar
Default: `'350.89198226'`

Equatorial radius — Equatorial radius

`3396200` (default) | double scalar

Equatorial radius for a custom central body, specified as a double scalar.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Gravitational potential model** to `None`, `Point-mass`, or `Oblate ellipsoid (J2)`.

Programmatic Use
Block Parameter: `customR`
Type: character vector
Values: `'3396200'` | double scalar
Default: `'3396200'`

Flattening — Flattening ratio

`0.00589` (default) | double scalar

Flattening ratio for custom central body, specified as a double scalar.

Dependencies

To enable this parameter, set:

- **Central body** to `Custom`.
- **Gravitational potential model** to `Point-mass`, `Oblate ellipsoid (J2)`, or `Spherical harmonics`.

Programmatic Use
Block Parameter: `customF`
Type: character vector

Values: `'0.00589'` | double scalar
Default: `'0.00589'`

Gravitational parameter — Gravitational parameter

`4.305e13` (default) | double scalar

Gravitational parameter for a custom central body, specified as a double scalar.

Dependencies

To enable this parameter, set:

- **Central body** to `Custom`.
- **Gravitational potential model** to `None`, `Point-mass`, or `Oblate ellipsoid (J2)`.

Programmatic Use
Block Parameter: `customMu`
Type: character vector
Values: `'4.305e13'` | double scalar
Default: `'4.305e13'`

Second degree zonal harmonic (J2) — Most significant or largest spherical harmonic term

`1.0826269e-03` (default) | double scalar

Most significant or largest spherical harmonic term, which accounts for oblateness of a celestial body, specified as a double scalar.

Dependencies

To enable this parameter, set:

- **Propagation method** to `Numerical (high precision)`.
- **Central body** to `Custom`.
- **Gravitational potential model** to `Oblate ellipsoid (J2)`.

Programmatic Use
Block Parameter: `customJ2`
Type: character vector
Values: `'1.0826269e-03'` | double scalar
Default: `'1.0826269e-03'`

Drag

Configure the atmospheric drag environment.

Include atmospheric drag — Option to include atmospheric drag

off (default) | on

To include atmospheric drag, select this check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.

Programmatic Use
Block Parameter: `useDrag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Atmospheric density source — Source of atmospheric density value

`Dialog` (default) | `Port`

Source of atmospheric density value, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `atmosSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Atmospheric model — Atmospheric model

`NRLMSISE-00` (default)

Atmospheric model for atmospheric drag calculation, specified as `NRLMSISE-00`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `atmosModel`
Type: character vector
Values: `'NRLMSISE-00'`
Default: `'NRLMSISE-00'`

Flux source — Source of space weather data

`Dialog` (default) | `Port`

Source for historical and predicted flux and geomagnetic indices used by atmospheric density calculation, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Programmatic Use

Block Parameter: `fluxSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Space weather data file — MAT-file of space weather data

`aeroSpaceWeatherData.mat` (default) | MAT-file name

MAT-file of space weather data. This file is the output from the `aeroReadSpaceWeatherData` function.

Aerospace Blockset includes a default space weather data file, `aeroSpaceWeatherData.mat`. To use the most recent data available, use `aeroReadSpaceWeatherData` to generate a new MAT-file, and specify the file name for this parameter. For more information, see `aeroReadSpaceWeatherData`.

If the file is not on the MATLAB path, specify the full pathname. The MAT-file must contain these variables from the space weather data file:

- `YEAR`
- `MONTH`
- `DAY`
- `AP1`
- `AP2`
- `AP3`
- `AP4`
- `AP5`
- `AP6`
- `AP7`
- `AP8`
- `AP_AVG`
- `F107_OBS`
- `F107_DATA_TYPE`
- `F107_OBS_CENTER81`

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `SpaceWeatherDataFile`
Type: character vector
Values: `'aeroSpaceWeatherData.mat'` | MAT-file name
Default: `'aeroSpaceWeatherData.mat'`

F10.7 extrapolation method — Extrapolation method for `f107Average` and `f107Daily`

`None - clip` (default) | `Constant` | `Least squares fit`

Extrapolation method for `f107Average` and `f107Daily` for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`None - clip`	Set `f107Average` and `f107Daily` to the nearest data point available in the MAT-file.
`Constant`	Set `f107Average` and `f107Daily` to a constant value specified by the **Magnetic index extrapolation method** parameter value.
`Least squares fit`	Approximate `f107Average` and `f107Daily` using a least-squares fit of the space weather data from October 1, 1957, to December 1, 2040. This method uses a trigonometric function of the form: `a + b*cos(c*t + d*sin(e*t))`, where: • *a* is `128.2351780622538` • *b* is `54.3285872213434` • *c* is `0.0015460708364` • *d* is `0.2429462096495` • *e* is `0.0015563188188` • *t* is *jdCurrent - jdReftime* • *jdCurrent* is the current Julian date • *jdRef* is `2436112.5`

Note For `f107Average` values,if part of the 81-day period falls inside the MAT-file range of space weather data, the Solar Flux and Geomagnetic Index block uses the actual daily values from the overlapping portion. To calculate the average for the nonoverlapping portion, the block uses the clipped, constant, or least squares daily value.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use
Block Parameter: `F107ExtrapMethod`
Type: character vector
Values: `'None - clip'` | `'Constant'` | `'Least squares fit'`
Default: `'None - clip'`

F10.7 extrapolation value — Extrapolation value to assign to `f107Average` and to calculate `f107Daily`

`150.0` (default) | scalar

Extrapolation value to assign to `f107Average` and to calculate `f107Daily`, specified as a scalar.

Tunable: Yes

Dependencies

This value is assigned when:

- **F10.7 extrapolation method** is set to `Constant`.
- Time specified by the **Year**, **DOY**, and **UT (sec)** ports is outside the range of the data in the MAT-file.

Programmatic Use
Block Parameter: `F107ExtrapValue`
Type: character vector
Values: `'150.0'` | scalar
Default: `'150.0'`

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

- Set **Atmospheric model** to `NRLMSISE-00`.
- Set **F10.7 extrapolation method** to `Constant`.

Magnetic index extrapolation method — Extrapolation method for magnetic index information

`None - clip` (default) | `Constant` | `IGRF`

Extrapolation method for magnetic index information values for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`None - clip`	Set elements of magnetic index information to the nearest data point available in the MAT-file.
`Constant`	Set elements of magnetic index information to a constant value specified by the **Magnetic index extrapolation method** parameter. The elements of magnetic index for times outside this range are based on clipped values of the horizontal magnetic field strength at these time limits.
`IGRF`	Calculate the elements of magnetic index information using the International Geomagnetic Reference Field. Because this model is defined for times between January 1, 1900, 12:00 AM UTC and January 1, 2025, 12:00 AM UTC, the predictions for times outside this range are clipped to values at these time limits.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `MagneticIndexExtrapMethod`
Type: character vector
Values: `'None - clip'` | `'Constant'` | `'IGRF'`
Default: `'None - clip'`

Magnetic index extrapolation value — Extrapolation value used to calculate magnetic index elements

`4.0` (default) | scalar

Extrapolation value used to calculate magnetic index elements, specified as a scalar.

Tunable: Yes

Dependencies

This parameter is enabled when **Magnetic index extrapolation method** is set to `Constant`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `MagneticIndexExtrapValue`
Type: character vector
Values: `'4.0'` | scalar
Default: `'4.0'`

Flags source — Variation flag source

`Dialog` (default) | `Port`

Variation flag source, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Atmospheric model** parameter to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `fluxFlagsSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Flags — Variation flags

`ones(1,23)` (default)

Variation flags, specified as an array of 23 (`ones(1,23)`). You can specify one of the following values for a field. The default value for each field is `1`.

- 0.0 — Removes the effect on the output.
- 1.0 — Applies the main and the cross-term effects of that value on the output.

- 2.0 — Applies only the cross-term effect of that value on the output.

The array has these fields.

Field	Description
`Flags(1)`	F10.7 effect on mean
`Flags(2)`	Independent of time
`Flags(3)`	Symmetrical annual
`Flags(4)`	Symmetrical semiannual
`Flags(5)`	Asymmetrical annual
`Flags(6)`	Asymmetrical semiannual
`Flags(7)`	Diurnal
`Flags(8)`	Semidiurnal
`Flags(9)`	Daily AP. If you set this field to -1, the block uses the entire matrix of magnetic index information (APH) instead of `APH(:,1)`.
`Flags(10)`	All UT, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Flags source** to `Dialog`.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Atmospheric model** parameter to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `fluxFlags`

Type: character vector
Values: `'ones(1,23)'`
Default: `'ones(1,23)'`

Include anomalous oxygen in density calculation — Option to include anomalous oxygen in density calculation

off (default) | on

To include anomalous oxygen in density calculations, select this check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Atmospheric model** parameter to `NRLMSISE-00`.

Programmatic Use
Block Parameter: `useOxygen`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Drag coefficient source — Source of drag coefficient

`Dialog` (default) | `Port`

Source of drag coefficient, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `dragCoeffSrc`
Type: character vector
Values: `'Dialog'` | `'Source'`
Default: `'Dialog'`

Drag coefficient — Spacecraft coefficient of drag

2.179 (default) | scalar | vector of size *numSat*

Spacecraft coefficient of drag used by atmospheric drag calculation, specified as a scalar or as a vector of size *numSat*.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag coefficient source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `dragCoeff`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'2.179'`

Drag area source — Source of drag area

`Dialog` (default) | `Port`

Source of drag area, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `dragAreaSrc`
Type: character vector
Values: `'Dialog'` | `'Source'`
Default: `'Dialog'`

Drag area — Area to compute acceleration due to atmospheric drag

1.0 (default) | scalar | vector of size *numSat*

Area to compute acceleration due to atmospheric drag, specified as a scalar or as a vector of size *numSat*. This area of the spacecraft is perpendicular to the spacecraft relative velocity.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag area source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `dragArea`

Type: character vector
Values: scalar | vector of size *numSat*
Default: '1.0'

Third Body

Configure third body point mass for gravitational acceleration.

Ephemeris model — Ephemeris model

DE405 (default) | DE421 | DE423 | DE430 | DE432t

Select one of these ephemerides models defined by the Jet Propulsion Laboratory. The block uses ephemeris data to calculate relative celestial positions required for third body point mass gravity and solar radiation pressure.

Ephemeris Model	Description
DE405	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE421	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE423	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE430	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE432t	Released in April 2014. This ephemeris takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. in the MATLAB desktop toolstrip, click **Add-Ons** .

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use

Block Parameter: `ephemerisModel`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Limit ephemerides date range — Option to enable start and end of range of ephemeris data

`off` (default) | `on`

Control how much data is loaded into memory during simulation and how much data is included in generated code for the block:

- Select this check box to limit the loading of ephemeris data to a specified date range.
- Clear this check box to include data for the complete date range defined in the **"Ephemeris model" on page 5-0** table.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use

Block Parameter: `useDateRange`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Start date (JD) — Start date of ephemerides date range

`juliandate(2020, 1, 1)` (default) | Julian date

Start date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use

Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1)'` | Julian date
Default: `'juliandate(2020, 1, 1)'`

End date (JD) — End date of ephemerides date range `juliandate(2050, 1, 1)` (default) | Julian date

End date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use

Block Parameter: `endDate`
Type: character vector
Values: `'juliandate(2050, 1, 1)'` | Julian format date
Default: `'juliandate(2050, 1, 1)'`

Include third body point-mass gravity — Acceleration due to third body gravity

`off` (default) | `on`

To include acceleration due to third body gravity in orbit propagation calculation, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use

Block Parameter: `useThirdBodyGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Sun — Gravitational acceleration due to Sun

`on` (default) | `off`

To include gravitational acceleration due to the Sun (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Sun`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeSunGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Mercury — Gravitational acceleration due to Mercury

`off` (default) | `on`

To include gravitational acceleration due to the Mercury (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Mercury`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeMercuryGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Venus — Gravitational acceleration due to Venus

`off` (default) | `on`

To include gravitational acceleration due to the Venus (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Venus`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeVenusGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Earth — Gravitational acceleration due to Venus

`on` (default) | `off`

To include gravitational acceleration due to the Earth (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Earth`.
- Set the **Include third body point-mass gravity** to `on`.

Programmatic Use
Block Parameter: `includeVenusGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Moon — Gravitational acceleration due to Moon

`on` (default) | `off`

To include gravitational acceleration due to the Moon (point mass), select this check box. Otherwise, clear it.

Programmatic Use
Block Parameter: `includeMoonGravity`
Type: character vector
Values: `'off'` | `'on'` |
Default: `'on'`

Mars — Gravitational acceleration due to Mars

`off` (default) | `on`

To include gravitational acceleration due to the Mars (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Mars`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeMarsGravity`
Type: character vector
Values: `'off'` | `'on'` |
Default: `'off'`

Jupiter — Gravitational acceleration due to Jupiter

`off` (default) | `on`

To include gravitational acceleration due to the Jupiter (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Jupiter`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includeJupiterGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Saturn — Gravitational acceleration due to Saturn

`off` (default) | `on`

To include gravitational acceleration due to the Saturn (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Saturn`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includeSaturnGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Uranus — Gravitational acceleration due to Uranus

`off` (default) | `on`

To include gravitational acceleration due to the Uranus (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Uranus`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includeUranusGravity`
Type: character vector
Values: `'off'` | `'on'`

Default: `'off'`

Neptune — Gravitational acceleration due to Neptune

`off` (default) | `on`

To include gravitational acceleration due to the Neptune (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Neptune`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includeNeptuneGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Pluto — Gravitational acceleration due to Pluto

`off` (default) | `on`

To include gravitational acceleration due to the Pluto (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Pluto`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includePlutoGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Custom — Gravitational acceleration due to custom planet

`off` (default) | `on`

To include gravitational acceleration due to a custom planet (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeCustomGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Custom gravitational parameter (m^3/s^2) — Custom gravitational acceleration

`42.828314258067e12` (default) | scalar | vector of length *numCustom3rdBodies*

Custom gravitational acceleration for custom third body, specified as a scalar or as a vector of length *numCustom3rdBodies*. *numCustom3rdBodies* is the number of rows provided to the **R_cb** input port. Provide more than one value to include multiple custom bodies.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include third body point-mass gravity** parameter to `on`.
- Set the **Custom** parameter to `on`.

Programmatic Use
Block Parameter: `customThirdBodyMu`
Type: character vector
Values: `'42.828314258067e12'` | scalar | vector of length *numCustom3rdBodies*
Default: `'42.828314258067e12'`

SRP

Include solar radiation pressure (SRP) — Acceleration due to solar radiation pressure

`off` (default) | `on`

To include acceleration due to solar radiation pressure in orbit propagation calculation, select this check box. Otherwise, clear the check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `useSRP`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Eclipse fraction source — Source of eclipse fraction

`Dialog` (default) | `Port`

Source of the eclipse fraction (fraction of solar disk visible from the spacecraft location), specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.

Programmatic Use
Block Parameter: `useSRP`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Shadow model — Shadow model

`Dual cone` (default) | `Cylindrical`

Shadow model for eclipse calculations, specified as one of these values.

- `Cylindrical` — Fraction can be 0.0 (Umbra) or 1.0 (Sunlight).
- `Dual cone` — Fraction can be 0.0 (Umbra), between 0.0 and 1.0 (Penumbra or Antumbra), or 1.0 (Sunlight).

Programmatic Use
Block Parameter: `shadowModel`
Type: character vector
Values: `'Dual cone'` | `'Cylindrical'`
Default: `'Dual cone'`

Include Earth — Option to include Earth

`on` (default) | `off`

Option to include Earth as a secondary occulting body in eclipse calculations when central body is Moon.

Dependencies

Set the **Central body** parameter to `Moon`.

Programmatic Use
Block Parameter: `includeEarth`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Include Moon — Option to include Moon

`on` (default) | `off`

Option to include Moon as a secondary occulting body in eclipse calculations when central body is Earth.

Dependencies

Set the **Central body** parameter to `Earth`.

Programmatic Use
Block Parameter: `includeMoon`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Reflectivity coefficient source — Source for spacecraft coefficient of reflectivity

`'Dialog'` (default) | `'Port'`

Source for the spacecraft coefficient of reflectivity, specified as `Dialog` or `Port`.

Dependencies

To enable this check box:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.

Programmatic Use
Block Parameter: `reflectivityCoeffSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'on'`

Reflectivity coefficient — Spacecraft coefficient of reflectivity

`1.8` (default) | scalar | 2D array of size *numSat* | between `[1,2]`

Spacecraft coefficient of reflectivity used by solar radiation pressure calculation, specified as a scalar, 2D array of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.
- Set the **Reflectivity coefficient source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `reflectivityCoeff`
Type: character vector
Values: `'1.8'` | scalar | 2D array of size *numSat* | between `[1,2]`
Default: `'1.8'`

SRP area source — Source for spacecraft solar radiation pressure area

`Dialog` (default) | `Port`

Source for the spacecraft solar radiation pressure (SRP) area, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.

Programmatic Use

Block Parameter: `srpAreaSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

SRP area — Cross section area of the spacecraft seen by the Sun

`1.0` (default) | scalar | 2D array of size *numSat*

Cross section area of the spacecraft seen by the Sun, specified as a scalar or as a 2D array of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.
- Set the **SRP area source** parameter to `Dialog`.

Programmatic Use

Block Parameter: `srpArea`
Type: character vector
Values: `'1.0'` | scalar | 2D array of size *numSat*
Default: `'1.0'`

Solar flux pressure (W*s/m^3) — Spacecraft solar flux pressure

`4.5344321e-6` (default) | scalar

Solar flux pressure, specified as a scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.

Programmatic Use

Block Parameter: `fluxPressure`

Type: character vector
Values: `4.5344321e-6` | scalar
Default: `'4.5344321e-6'`

Units

Units — Parameter and port units

`Metric (m/s)` (default) | `Metric (km/s)` | `Metric (km/h)` | `English (ft/s)` | `English (kts)`

Parameter and port units, specified as:

Units	Distance	Velocity	Acceleration	Mass	Area	Density
`Metric (m/s)`	meters	meters/sec	meters/sec^2	Kilograms	m^2	kg/m^3, some density outputs 1/m^3
`Metric (km/s)`	kilometers	kilometers/sec	kilometers/sec^2	Kilograms	m^2	kg/m^3, some density outputs 1/m^3
`Metric (km/h)`	kilometers	kilometers/hour	kilometers/hour^2	Kilograms	m^2	kg/m^3, some density outputs 1/m^3
`English (ft/s)`	feet	feet/sec	feet/sec^2	Slugs	feet^2	lbm/ft^3, some density outputs 1/ft^3
`English (kts)`	nautical mile	knots	knots/sec	Slugs	feet^2	lbm/ft^3, some density outputs 1/ft^3

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (m/s)'` | `'Metric (km/s)'` | `'Metric (km/h)'` | `'English (ft/s)'` | `'English (kts)'`
Default: `'Metric (m/s)'`

Angle units — Angle units

`Degrees` (default) | `Radians`

Parameter and port units for angles, specified as `Degrees` or `Radians`.

Programmatic Use
Block Parameter: `angleUnits`
Type: character vector
Values: `'Degrees'` | `'Radians'`
Default: `'Degrees'`

Time format — Time format for start date and time output

`Julian date` (default) | `Gregorian`

Time format for **Start date/time (UTC Julian date)** and output port $\mathbf{t_{utc}}$, specified as `Julian date` or `Gregorian`.

Programmatic Use
Block Parameter: `timeFormat`
Type: character vector
Values: `'Julian date'` | `'Gregorian'`
Default: `'Julian date'`

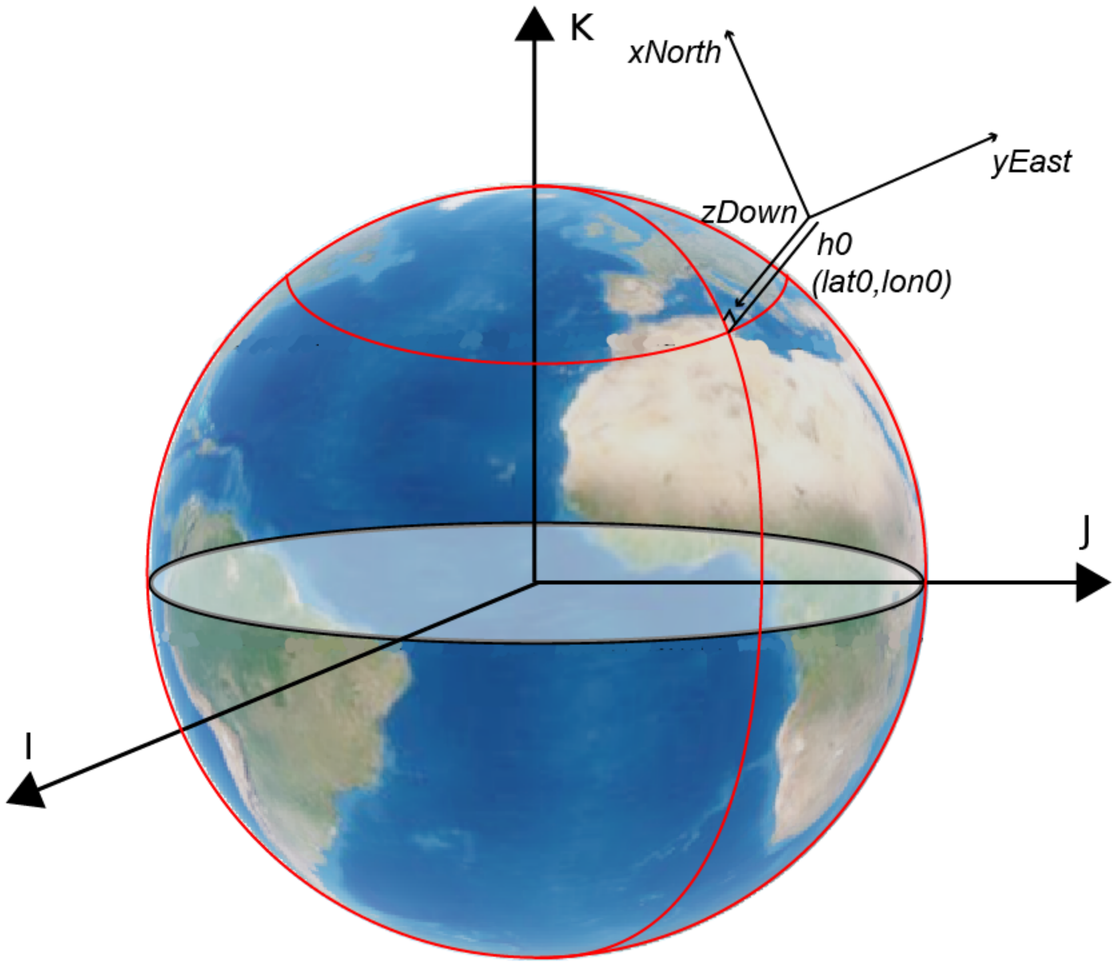
Algorithms

Coordinate Systems

The Orbit Propagator block works in the ICRF and fixed-frame coordinate systems:

- *ICRF* — International Celestial Reference Frame (ICRF). This frame can be treated as equal to the ECI coordinate system realized at J2000 (Jan 1 2000 12:00:00 TT. For more information, see “ECI Coordinates” on page 2-12).

  To describe a point in space, you need a frame of reference that does not rotate with respect to the stars. The ICRF, with the origin at the center of the Earth and orthogonal vectors **I**, **J**, and **K**, is used as the frame of reference. The fundamental plane is the **IJ**-plane, which is closely aligned with the equator with a small offset that changes over time because of precession and nutation of the rotation axis of the Earth.



- *Fixed-frame* — Fixed-frame is a generic term for the coordinate system that is fixed to the central body (its axes rotate with the central body and are not fixed in inertial space). For high precision orbit propagation methods
  - When the central body is earth, and the Earth orientation parameters (EOPs) are used, the Fixed-frame for Earth is the International Terrestrial Reference Frame (ITRF). This reference frame is realized by the IAU2000/2006 reduction from the ICRF coordinate system using the earth orientation parameter file provided. If Earth orientation parameters are not used, the block still uses the IAU2000/2006 reduction, but with Earth orientation parameters set to `0`.
  - When the central body is moon, and the moon libration angles are provided as input, the fixed-frame coordinate system for the moon is the Mean Earth/pole axis frame (ME). This frame is realized by two transformations. First, the values in the ICRF frame are transformed into the Principal Axis system (PA), the axis defined by the libration angles provided as inputs to the block. For more information, see Moon Libration. The states are then transformed into the ME system using a fixed rotation from the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [5]. If the Moon libration angles input is not provided, the fixed frame is defined by the directions of the poles of rotation and prime meridians defined in the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [5].

- When the Central Body is custom, the fixed-frame coordinate system is defined by the poles of rotation and prime meridian defined by the block input α, δ, W, or the spin axis properties.

In all other cases, the fixed frame for each central body is defined by the directions of the poles of rotation and prime meridians defined in the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [5].

Orbit Propagation Methods

The Aerospace Blockset supports two top-level orbit propagation methods: `Kepler (unperturbed)` and `Numerical (high precision)`.

For more information, see “Orbit Propagation Methods” on page 2-46.

Version History

Introduced in R2020b

R2023b: Orbit Propagator Block Changes

The Orbit Propagator block has been updated to take into account:

- The effects of third body gravity on orbit propagation.
- Space weather data from a data file generated by the `aeroReadSpaceWeatherData` function.
- Solar radiation pressure (SRP).

References

[1] Vallado, David. *Fundamentals of Astrodynamics and Applications*, 4th ed. Hawthorne, CA: Microcosm Press, 2013.

[2] Gottlieb, R. G., "Fast Gravity, Gravity Partials, Normalized Gravity, Gravity Gradient Torque and Magnetic Field: Derivation, Code and Data," Technical Report NASA Contractor Report 188243, NASA Lyndon B. Johnson Space Center, Houston, Texas, February 1993.

[3] Konopliv, A. S., S. W. Asmar, E. Carranza, W. L. Sjogen, D. N. Yuan., "Recent Gravity Models as a Result of the Lunar Prospector Mission, Icarus", Vol. 150, no. 1, pp 1–18, 2001.

[4] Lemoine, F. G., D. E. Smith, D.D. Rowlands, M.T. Zuber, G. A. Neumann, and D. S. Chinn, "An improved solution of the gravity field of Mars (GMM-2B) from Mars Global Surveyor", Journal Of Geophysical Research, Vol. 106, No. E10, pp 23359-23376, October 25, 2001.

[5] Seidelmann, P.K., Archinal, B.A., A’hearn, M.F. et al. "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006." *Celestial Mech Dyn Astr* 98, 155–180 (2007).

[6] Montenbruck, Oliver, and Gill Eberhard. Satellite Orbits: Models, Methods, and Applications. Springer, 2000.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

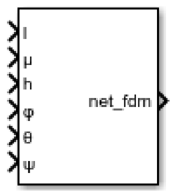
Blocks
CubeSat Vehicle | Moon Libration | Attitude Profile | Solar Flux and Geomagnetic Index | Spacecraft Dynamics | NRLMSISE-00 Atmosphere Model

Functions
`aeroReadSpaceWeatherData`

Topics
"Model and Simulate CubeSats" on page 2-73

Pack net_fdm Packet for FlightGear

Generate `net_fdm` packet for FlightGear



Libraries:
Aerospace Blockset / Animation / Flight Simulator Interfaces

Description

The Pack net_fdm Packet for FlightGear block creates, from separate inputs, a FlightGear `net_fdm` data packet compatible with a particular version of FlightGear flight simulator. This block accepts all signals supported by the FlightGear `net_fdm` data packet. These signals are arranged into six groups:

- Position/attitude inputs
- Velocity/acceleration inputs
- Control surface position inputs
- Engine/fuel inputs
- Landing gear inputs
- Environment inputs

To enable or disable the inputs for these groups, select the associated block parameter. The block input ports change depending on the requested signal groups. The block inserts zeros for packet values that are part of inactive signal groups.

The Aerospace Blockset product supports FlightGear versions starting from v2.6. If you are using a FlightGear version older than 2.6, the model displays a notification from the Simulink Upgrade Advisor. Consider using the Upgrade Advisor to upgrade your FlightGear version. For more information, see "Supported FlightGear Versions" on page 2-20.

Ports

Input

Position/Attitude Inputs

l — Longitude
scalar

Longitude, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show position/attitude inputs** check box.

Data Types: `double`

μ — Latitude
scalar

Latitude, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show position/attitude inputs** check box.

Data Types: `double`

h — Altitude
scalar

Altitude, specified as a scalar, in m.

Dependencies

To enable this port, select the **Show position/attitude inputs** check box.

Data Types: `double`

ϕ — Roll
scalar

Roll, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show position/attitude inputs** check box.

Data Types: `single`

θ — Pitch
scalar

Pitch, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show position/attitude inputs** check box.

Data Types: `single`

ψ — Yaw
scalar

Yaw, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show position/attitude inputs** check box.

Data Types: `single`

Velocity/Acceleration Inputs

α — Angle of attack
scalar

Angle of attack, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

β — Sideslip angle
scalar

Sideslip angle, specified as a scalar, in rad.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

dφ/dt — Roll rate
scalar

Roll rate, specified as a scalar, in rad/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

dθ/dt — Pitch rate
scalar

Pitch rate, specified as a scalar, in rad/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

dψ/dt — Yaw rate
scalar

Yaw rate, specified as a scalar, in rad/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

V_{cas} — Calibrated airspeed
scalar

Calibrated airspeed, specified as a scalar, in knots.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

climb_rate — Rate of climb
scalar

Rate of climb, specified as a scalar, in feet/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

v_{north} — North velocity in body frame
scalar

North velocity in body frame, specified as a scalar, in ft/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

v_{east} — East velocity in body frame
scalar

East velocity in body frame, specified as a scalar, in feet/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

v_{down} — Down velocity
scalar

Down velocity, specified as a scalar, in feet/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

$v_{wind\ body\ north}$ — North velocity in body frame relative to local airmass
scalar

North velocity in body frame relative to local airmass, specified as a scalar, in ft/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

$v_{wind\ body\ east}$ — East velocity in body frame relative to local airmass
scalar

East velocity in body frame relative to local airmass, specified as a scalar, in ft/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

$v_{wind\ body\ down}$ — Down velocity in body frame relative to airmass
scalar

Down velocity in body frame relative to airmass, specified as a scalar, in ft/sec.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

$A_{X\ pilot}$ — *X* acceleration in body frame
scalar

X acceleration in body frame, specified as a scalar, in ft/sec^2.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

$A_{Y\ pilot}$ — *Y* acceleration in body frame
scalar

Y acceleration in body frame, specified as a scalar, in ft/sec^2.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

$A_{Z\ pilot}$ — *Z* acceleration in body frame
scalar

Z acceleration in body frame, specified as a scalar, in ft/sec^2.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

stall_warning — Amount of stall
scalar

Amount of stall [0-1], specified as a scalar.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

slip_degree — Slip ball deflection
scalar

Slip ball deflection, specified as a scalar, in degrees.

Dependencies

To enable this port, select the **Show velocity/acceleration inputs** check box.

Data Types: `single`

Control Surface Position Inputs

elevator — Normalized elevator position
scalar

Normalized elevator position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

elevator_trim_tab — Normalized elevator trim tab position
scalar

Normalized elevator trim tab position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

left_flap — Normalized left flap position
scalar

Normalized left flap position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

right_flap — Normalized right flap position
scalar

Normalized right flap position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

left_aileron — Normalized left aileron position
scalar

Normalized left aileron position. specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

right_aileron — Normalized right aileron position
scalar

Normalized right aileron position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

rudder — Normalized rudder position
scalar

Normalized rudder position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

nose_wheel — Normalized nose wheel position
scalar

Normalized nose wheel position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

speedbrake — Normalized speedbrake position
scalar

Normalized speedbrake position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

spoilers — Normalized spoilers position
scalar

Normalized spoilers position, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position inputs** check box.

Data Types: `single`

Engine/Fuel Inputs

num_engines — Number of engines
scalar

Number of engines, specified as a scalar.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `uint32`

eng_state — Engine state
vector

Engine state (off, cranking, running), specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `uint32`

rpm — Engine RPM
vector

Engine RPM, specified as a vector, in rev/min.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

fuel_flow — Fuel flow
vector

Fuel flow, specified as a vector, in gal/hr.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

fuel_px — Fuel pressure
vector

Fuel pressure, specified as a vector, in psi.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

egt — Exhaust gas temperature
vector

Exhaust gas temperature, specified as a vector, in deg F.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

cht — Cylinder head temperature
scalar

Cylinder head temperature, specified as a vector, in deg F.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

mp_osi — Manifold pressure
vector

Manifold pressure, specified as a vector, in psi.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

tit — Turbine inlet temperature
vector

Turbine inlet temperature, specified as a vector, in deg F.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

oil_temp — Oil temperature
vector

Oil temperature, specified as a vector, in deg F.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

oil_px — Oil pressure
vector

Oil pressure, specified as a vector, in psi.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

num_tanks — Number of fuel tanks
scalar

Number of fuel tanks, specified as a scalar.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `uin32`

fuel_quantity — Fuel quantity per tank
vector

Fuel quantity per tank, specified as a vector, in gal.

Dependencies

To enable this port, select the **Show engine/fuel inputs** check box.

Data Types: `single`

Landing Gear Inputs

num_wheels — Number of wheels
scalar

Number of wheels, specified as a scalar.

Dependencies

To enable this port, select the **Show landing gear inputs** check box.

Data Types: `uint32`

wow — Weight on wheels switch
vector

Weight on wheels switch, specified as a vector.

Dependencies

To enable this port, select the **Show landing gear inputs** check box.

Data Types: `uint32`

gear_pos — Landing gear normalized position
vector

Landing gear normalized position, specified as a vector.

Dependencies

To enable this port, select the **Show landing gear inputs** check box.

Data Types: `single`

gear_steer — Landing gear normalized steering
vector

Landing gear normalized steering, specified as a vector.

Dependencies

To enable this port, select the **Show landing gear inputs** check box.

Data Types: `single`

gear_compression — Landing gear normalized compression
vector

Landing gear normalized compression, specified as a vector.

Dependencies

To enable this port, select the **Show landing gear inputs** check box.

Data Types: `single`

Environment Inputs

agl — Altitude above ground level
scalar

Altitude above ground level, specified as a scalar, in m.

Dependencies

To enable this port, select the **Show environment inputs** check box.

Data Types: `single`

cur_time — Current UNIX® time
scalar

Current UNIX time, specified as a scalar, in sec.

Dependencies

To enable this port, select the **Show environment inputs** check box.

Data Types: `uint32`

warp — Offset in seconds to UNIX time
scalar

Offset in seconds to UNIX time, specified as a scalar, in sec.

Dependencies

To enable this port, select the **Show environment inputs** check box.

Data Types: `int32`

visibility — Visibility
scalar

Visibility (for visual effects), specified as a scalar, in m.

Dependencies

To enable this port, select the **Show environment inputs** check box.

Data Types: `single`

Output

net_fdm — Packet generated for FlightGear
array

Packet generated for FlightGear, specified as an array.

Data Types: `single` | `double` | `uint32`

Parameters

Show position/altitude inputs — Position and altitude inputs

`on` (default) | `off`

Select this check box to include the position and altitude inputs in the FlightGear `net_fdm` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 1: Position/Altitude Inputs

Name	Units	Type	Width	Description
longitude	rad	double	1	Geodetic longitude
latitude	rad	double	1	Geodetic latitude
altitude	m	double	1	Altitude above sea level
theta	rad	single	1	Pitch
phi	rad	single	1	Roll
psi	rad	single	1	Yaw

Programmatic Use
Block Parameter: `ShowPositionAttitudeInputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

Show velocity/acceleration inputs — Velocity and acceleration inputs

`off` (default) | `on`

Select this check box to include the velocity and acceleration inputs in the FlightGear `net_fdm` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 2: Velocity/Acceleration Inputs

Name	Units	Type	Width	Description
alpha	rad	double	1	Angle of attack
beta	rad	single	1	Sideslip angle
dphi/dt	rad/sec	single	1	Roll rate
dtheta/dt	rad/sec	single	1	Pitch rate
dpsi/dt	rad/sec	single	1	Yaw rate
Vcas	knot	single	1	Calibrated airspeed
climb_rate	feet/sec	single	1	Rate of climb
v_north	feet/sec	single	1	North velocity in body frame
v_east	feet/sec	single	1	East velocity in body frame
v_down	feet/sec	single	1	Down velocity
v_wind_body_north	feet/sec	single	1	North velocity in body frame relative to local airmass
v_wind_body_east	feet/sec	single	1	East velocity in body frame relative to local airmass
v_wind_body_down	feet/sec	single	1	Down velocity in body frame relative to airmass
Axpilot	feet/sec^2	single	1	X acceleration in body frame
Aypilot	feet/sec^2	single	1	Y acceleration in body frame
Azpilot	feet/sec^2	single	1	Z acceleration in body frame
stall_warning	—	single	1	Amount of stall [0-1]
slip_deg	degrees	single	1	Slip ball deflection

Programmatic Use
Block Parameter: `ShowVelocityAccelerationInputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Show control surface position inputs — Control surface position inputs

`off` (default) | `on`

Select this check box to include the control surface position inputs in the FlightGear `net_fdm` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 3: Control Surface Position Inputs

Name	Units	Type	Width	Description
elevator	1 (dimensionless)	single	1	Normalized elevator position
elevator_trim_tab	1 (dimensionless)	single	1	Normalized elevator trim tab position
left_flap	1 (dimensionless)	single	1	Normalized left flap position
right_flap	1 (dimensionless)	single	1	Normalized right flap position
left_aileron	1 (dimensionless)	single	1	Normalized left aileron position
right_aileron	1 (dimensionless)	single	1	Normalized right aileron position
rudder	1 (dimensionless)	single	1	Normalized rudder position
nose_wheel	1 (dimensionless)	single	1	Normalized nose wheel position
speedbrake	1 (dimensionless)	single	1	Normalized speedbrake position
spoilers	1 (dimensionless)	single	1	Normalized spoilers position

Programmatic Use
Block Parameter: `ShowControlSurfacePositionInputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Show engine/fuel inputs — Engine and fuel inputs

`off` (default) | `on`

Select this check box to include the engine and fuel inputs in the FlightGear `net_fdm` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 4: Engine/Fuel Inputs

Name	Units	Type	Width	Description
num_engines	—	uint32	1	Number of engines
eng_state	—	uint32	4	Engine state (off, cranking, running)
rpm	rev/min	single	4	Engine RPM
fuel_flow	gal/hour	single	4	Fuel flow
fuel_px	psi	single	4	Fuel pressure
egt	deg F	single	4	Exhaust gas temperature
cht	deg F	single	4	Cylinder head temperature
mp_osi	psi	single	4	Manifold pressure
tit	deg F	single	4	Turbine inlet temperature
oil_temp	deg F	single	4	Oil temperature
oil_px	psi	single	4	Oil pressure
num_tanks	—	uint32	1	Number of fuel tanks
fuel_quantity	gal	single	4	Fuel quantity per tank

Programmatic Use
Block Parameter: `ShowEngineFuelInputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Show landing gear inputs — Landing gear inputs

`off` (default) | `on`

Select this check box to include the landing gear inputs in the FlightGear `net_fdm` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 5: Landing Gear Inputs

Name	Units	Type	Width	Description
num_wheels	—	uint32	1	Number of wheels
wow	—	uint32	3	Weight on wheels switch
gear_pos	—	single	3	Landing gear normalized position
gear_steer	—	single	3	Landing gear normalized steering
gear_compression	—	single	3	Landing gear normalized compression

Programmatic Use
Block Parameter: `ShowLandingGearInputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Show environment inputs — Environment inputs

`off` (default) | `on`

Select this check box to include the environment inputs in the FlightGear `net_fdm` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 6: Environment Inputs

Name	Units	Type	Width	Description
agl	m	single	1	Altitude above ground level
cur_time	sec	uint32	1	Current UNIX time
warp	sec	int32	1	Offset in seconds to UNIX time
visibility	m	single	1	Visibility in meters (for visual effects)

Programmatic Use
Block Parameter: `ShowEnvironmentInputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Sample time — Sample time

`1/30` (default) | scalar

Specify the sample time (-1 for inherited).

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'1/30'`

Version History

Introduced before R2006a

See Also

FlightGear Preconfigured 6DoF Animation | Generate Run Script | Receive net_ctrl Packet from FlightGear | Send net_fdm Packet to FlightGear | Unpack net_ctrl Packet from FlightGear

Topics

“Flight Simulator Interface” on page 2-20
“Work with the Flight Simulator Interface” on page 2-24

Pilot Joystick

Provide joystick interface on Windows platform

Libraries:
Aerospace Blockset / Animation / Animation Support Utilities

Description

The Pilot Joystick block provides a pilot joystick interface for a Windows platform. Roll, pitch, yaw, and throttle are mapped to the joystick *X*, *Y*, *R*, and *Z* channels respectively.

You can also configure the block to output all channels by setting the **Output configuration** parameter to `AllOutputs`. For more information, see Pilot Joystick All. The Pilot Joystick and Pilot Joystick All blocks are identical blocks with different **Output configuration** default settings.

This block does not produce deployable code.

Limitations

- The Pilot Joystick block is not supported in Simulink Online.
- If the joystick does not support an *R* (rudder or twist) channel, yaw output is set to zero. Outputs are of type double, except when **Joystick ID** is set to `AllOutputs` mode, which is a uint32 flagword of bits. On non Microsoft platforms, this block outputs zeros.
- Pitch value has the opposite sense as that delivered by the FlightGear joystick interface.

Ports

Output

roll — Roll
range [-1, 1]

Roll command, specified in the range [-1, 1], that corresponds to the joystick left and right directions.

Dependencies

To enable this port, set **Output configuration** to `FourAxis`.

Data Types: `double`

pitch — Pitch
range [-1, 1]

Pitch command, specified in the range [-1, 1], that corresponds to the joystick forward or down and back and up directions.

Dependencies

This output port is enabled when the **Output configuration** parameter is set to `FourAxis`.

Data Types: `double`

yaw — Yaw
range [-1, 1]

Yaw command, specified in the range [-1, 1], that corresponds to the joystick twist left and twist right directions.

Dependencies

To enable this port, set **Output configuration** to `FourAxis`.

Data Types: `double`

throttle — Throttle
range [0, 1]

Throttle command, specified in the range [0, 1], that corresponds to the joystick min and max position.

Dependencies

To enable this port, set **Output configuration** to `FourAxis`.

Data Types: `double`

Parameters

Joystick ID — Joystick ID

`Joystick1` (default) | `Joystick2` | `Joystick3` | `Joystick4` | `Joystick5` | `Joystick6` | `Joystick7` | `Joystick8` | `Joystick9` | `Joystick10` | `Joystick11` | `Joystick12` | `Joystick13` | `Joystick14` | `Joystick15` | `Joystick16` | `None`

Specify the joystick ID as

- `Joystick1`
- `Joystick2`
- `Joystick3`
- `Joystick4`
- `Joystick5`
- `Joystick6`
- `Joystick7`
- `Joystick8`
- `Joystick9`
- `Joystick10`
- `Joystick11`
- `Joystick12`

- `Joystick13`
- `Joystick14`
- `Joystick15`
- `Joystick16`
- `None`

Programmatic Use
Block Parameter: `JoystickID`
Type: character vector
Values: `Joystick1` | `Joystick2` | `Joystick3` | `Joystick4` | `Joystick5` | `Joystick6` | `Joystick7` | `Joystick8` | `Joystick9` | `Joystick10` | `Joystick11` | `Joystick12` | `Joystick13` | `Joystick14` | `Joystick15` | `Joystick16` | `None`
Default: `'Joystick1'`

Output configuration — Joystick output configuration

`FourAxis` (default) | `AllOutputs`

Joystick output configuration, specified as `FourAxis` or `AllOutputs`. For more information on the `AllOutputs` configuration, see Pilot Joystick All.

Programmatic Use
Block Parameter: `OutputConfiguration`
Type: character vector
Values: `FourAxis` | `AllOutputs`
Default: `'FourAxis'`

Sample time — Sample time

`1/30` (default) | `-1` | scalar

Specify the sample time (-1 for inherited), specified as a scalar.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'1/30'`

Version History

Introduced before R2006a

R2023b: Pilot Joystick Block Update

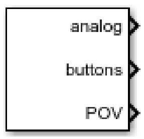
The Pilot Joystick block now supports up to 16 joysticks.

See Also

Pilot Joystick All | Simulation Pace

Pilot Joystick All

Provide joystick interface in All Outputs configuration on Windows platform



Libraries:
Aerospace Blockset / Animation / Animation Support Utilities

Description

The Pilot Joystick All block provides a pilot joystick interface for a Windows platform. Analog is mapped to the joystick X, Y, Z, R, U, and V channels. Buttons and POV are mapped to up to 32 joystick button states and the joystick point-of-view hat.

You can also configure the block to output four axes by setting the **Output configuration** parameter to `FourAxis`.

This block does not produce deployable code.

Limitations

- The Pilot Joystick All block is not supported in Simulink Online.
- If the joystick does not support an *R* (rudder or twist) channel, yaw output is set to zero. Outputs are of type double, except when **Joystick ID** is set to `AllOutputs` mode, which is a `uint32` flagword of bits. On non Microsoft platforms, this block outputs zeros.
- Pitch value has the opposite sense as that delivered by the FlightGear joystick interface.

Ports

Output

analog — Analog output
range [-1, 1] | range [0, 1]

Analog output, returned according to:

Array Number	Channel	Output Range	Joystick	Description
1	X	[-1, 1]	[left, right]	Roll command
2	Y	[-1, 1]	[forward/down, back/up]	Pitch command
3	Z	[0, 1]	[min, max]	Throttle command
4	R	[-1, 1]	[left, right]	Yaw command
5	U	[0, 1]	[min, max]	U channel value
6	V	[0, 1]	[min, max]	V channel value

Data Types: `double`

buttons — Button
flagword with 32 button states

Button output, returned as a flagword containing up to 32 button states on the buttons channel. Bit 0 is button 1, Bit 1 is button 2, and so forth.

Data Types: `uint32`

POV — Point-of-view
hat

Point-of-view, returned as a hat value in degrees on the POV channel. Zero degrees is straight ahead, 90 degrees is to the left, and so forth.

Data Types: `double`

Parameters

Joystick ID — Joystick ID

`Joystick1` (default) | `Joystick2` | `Joystick3` | `Joystick4` | `Joystick5` | `Joystick6` | `Joystick7` | `Joystick8` | `Joystick9` | `Joystick10` | `Joystick11` | `Joystick12` | `Joystick13` | `Joystick14` | `Joystick15` | `Joystick16` | `None`

Specify the joystick ID as

- `Joystick1`
- `Joystick2`
- `Joystick3`
- `Joystick4`
- `Joystick5`
- `Joystick6`
- `Joystick7`
- `Joystick8`
- `Joystick9`
- `Joystick10`
- `Joystick11`
- `Joystick12`
- `Joystick13`
- `Joystick14`
- `Joystick15`
- `Joystick16`
- `None`

Programmatic Use
Block Parameter: `JoystickID`
Type: character vector

Values: Joystick1 | Joystick2 | Joystick3 | Joystick4 | Joystick5 | Joystick6 | Joystick7 | Joystick8 | Joystick9 | Joystick10 | Joystick11 | Joystick12 | Joystick13 | Joystick14 | Joystick15 | Joystick16 | None
Default: 'Joystick1'

Output configuration — Joystick output configuration
AllOutputs (default) | FourAxis

Joystick output configuration, specified as FourAxis or AllOutputs. For more information on the AllOutputs configuration, see Pilot Joystick All.

Programmatic Use
Block Parameter: OutputConfiguration
Type: character vector
Values: FourAxis | AllOutputs
Default: 'FourAxis'

Sample time — Sample time
1/30 (default) | -1 | scalar

Specify the sample time (-1 for inherited), specified as a scalar.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '1/30'

Version History

Introduced in R2007a

R2023b: Pilot Joystick All Block Update

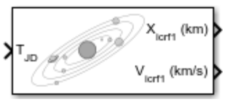
The Pilot Joystick All block now supports up to 16 joysticks.

See Also

Pilot Joystick | Simulation Pace

Planetary Ephemeris

Implement position and velocity of astronomical objects



Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Description

The Planetary Ephemeris block uses Chebyshev coefficients to implement the position and velocity of the target object relative to the specified center object for a given Julian date. The **Target** parameter specifies an astronomical object. The block implements the ephemerides using the **Center** parameter for an astronomical object as the reference.

The block uses the Chebyshev coefficients that the NASA Jet Propulsion Laboratory provides.

Tip For T_{JD}, Julian date input for the block:

- Calculate the date using the Julian Date Conversion block or the Aerospace Toolbox `juliandate` function.
- Calculate the Julian date using some other means and input it using the Constant block.

This block implements the position and velocity using the International Celestial Reference Frame. If you require the planetary ephemeris position value relative to Earth in Earth-fixed (ECEF) coordinates, use the Direction Cosine Matrix ECI to ECEF block.

Ports

Input

T_{JD} — Julian date
scalar | positive | between minimum and maximum Julian dates

Julian date, specified as a positive scalar between minimum and maximum Julian dates.

Specify the Julian dates in Barycentric Dynamical Time (TDB).

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `Julian date`.

Data Types: `double`

$T0_{JD}$ — Fixed Julian date
scalar | positive

Fixed Julian date for a specific epoch that is the most recent midnight at or before the interpolation epoch, specified as a positive scalar. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian date.

Specify the Julian dates in Barycentric Dynamical Time (TDB).

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `T0 and elapsed Julian time`.

Data Types: `double`

ΔT_{JD} — Elapsed Julian time
scalar | positive

Elapsed Julian time between the fixed Julian date and the ephemeris time, specified as a positive scalar. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian date.

See the **Ephemeris model** parameter for the minimum and maximum Julian dates.

Dependencies

This port displays if the **Epoch** parameter is set to `T0 and elapsed Julian time`.

Data Types: `double`

Output

X_{icrf1} — Barycenter position
vector

Barycenter position (X_{icrf1}) of the **Target** object relative to the barycenter of the **Center** object, output as a vector, in km or astronomical units (AU).

Tip This block outputs the barycenter position in International Celestial Reference Frame (ICRF) coordinates. To convert these coordinates to Earth-centered Earth-fixed (ECEF), use the Direction Cosine Matrix ECI to ECEF block.

Data Types: `double`

V_{icrf} — Velocity
vector

Velocity (V_{icrf}) of the barycenter of the **Target** object relative to the barycenter of the **Center** object, specified as a vector, in km/s or astronomical units (AU)/day.

Data Types: `double`

Parameters

Units — Output units

`km,km/s` (default) | `AU,AU/day`

Output units, specified as `km,km/s` or `AU,AU/day`.

Units	Position	Velocity
`m,m/s`	m	m/s
`km,km/s`	km	km/s
`Au,AU/day`	astronomical units (AU)	AU/day

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `m,m/s` | `km,km/s` | `AU,AU/day`
Default: `'km,km/s'`

Epoch — Epoch

`Julian date` (default) | `T0 and elapsed Julian time`

Epoch, specified as:

- `Julian date`

  Julian date to implement the position and velocity of the **Target** object.. When this option is selected, the block has one input port, T_{JD}.
- `T0 and elapsed Julian time`

  Julian date, specified by two block inputs:

  - Fixed Julian date representing a starting epoch.
  - Elapsed Julian time between the fixed Julian date ($T0_{JD}$) and the desired model simulation time. The sum of $T0_{JD}$ and ΔT_{JD} must fall between the minimum and maximum Julian date.

Programmatic Use
Block Parameter: `epochFormat`
Type: character vector
Values: `Julian date` | `T0 and elapsed Julian time`
Default: `'Julian date'`

Ephemeris model — Ephemeris model

`DE405` (default) | `DE421` | `DE423` | `DE430` | `DE432t`

Select one of these ephemerides models defined by the Jet Propulsion Laboratory. The block uses ephemeris data to calculate relative celestial positions required for third body point mass gravity and solar radiation pressure.

Ephemeris Model	Description
`DE405`	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.

Ephemeris Model	Description
DE421	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE423	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE430	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE432t	Released in April 2014. This ephemeris takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. in the MATLAB desktop toolstrip, click **Add-Ons** .

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `ephemerisModel`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Ephemeris model — Ephemeris model

`DE405` (default) | `DE421` | `DE423` | `DE430` | `DE432t`

Select one of these ephemerides models defined by the Jet Propulsion Laboratory.

Ephemeris Model	Description
DE405	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.

Ephemeris Model	Description
DE421	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE423	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE430	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE432t	Released in April 2014. This ephemeris takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. In the MATLAB desktop toolstrip, click **Add-Ons**.

Programmatic Use
Block Parameter: `ephemerisModel`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Center — Center body

Sun (default) | `Mercury` | `Venus` | `Earth` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Pluto` | `Solar system barycenter` | `Earth-Moon barycenter`

Center body (astronomical object) or reference body, specified as a point of reference for the **Target** barycenter position and velocity measurement.

Programmatic Use
Block Parameter: `center`
Type: character vector
Values: `Sun` | `Mercury` | `Venus` | `Earth` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Pluto` | `Solar system barycenter` | `Earth-Moon barycenter`
Default: `'Sun'`

Target — Target body

Sun (default) | `Mercury` | `Venus` | `Earth` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Pluto` | `Solar system barycenter` | `Earth-Moon barycenter`

Target body (astronomical object) or reference body, specified as a point of reference for the barycenter position and velocity measurement.

Programmatic Use
Block Parameter: `target`
Type: character vector
Values: `Sun` | `Mercury` | `Venus` | `Earth` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Pluto` | `Solar system barycenter` | `Earth-Moon barycenter`
Default: `'Moon'`

Limit ephemerides date range — Option to enable start and end of range of ephemeris data

`off` (default) | `on`

Control how much data is loaded into memory during simulation and how much data is included in generated code for the block:

- Select this check box to limit the loading of ephemeris data to a specified date range.
- Clear this check box to include data for the complete date range defined in the **"Ephemeris model" on page 5-0** table.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `useDateRange`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Start date (JD) — Start date of ephemerides date range

`juliandate(2020, 1, 1)` (default) | Julian date

Start date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use
Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1)'` | Julian date
Default: `'juliandate(2020, 1, 1)'`

End date (JD) — End date of ephemerides date range

`juliandate(2050, 1, 1)` (default) | Julian date

End date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use
Block Parameter: `endDate`
Type: character vector
Values: `'juliandate(2050, 1, 1)'` | Julian format date
Default: `'juliandate(2050, 1, 1)'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Calculate velocity — Calculate rate of target barycenter

`on` (default) | `off`

Select this check box to calculate the velocity of the **Target** barycenter relative to the **Center** barycenter.

Programmatic Use
Block Parameter: `outputVelocity`
Type: character vector
Values: `'off'` | `'on'` |
Default: `'on'`

Version History

Introduced in R2013a

R2023a: Planetary Ephemeris Block Controls Range of Ephemeris Data

Behavior changed in R2023a

You can now control the range of ephemeris data for the Planetary Ephemeris using these new parameters:

- **Limit ephemerides date range** — Select this check box to enable the start and end of the range of ephemeris data.
- **Start data** — Enter the start date of the range of ephemeris data. The **Epoch** parameter affects this date format.
- **End data** — Enter the end date of the range of ephemeris data. The **Epoch** parameter affects this date format.

References

[1] Folkner, W. M., J. G. Williams, and D. H. Boggs. "The Planetary and Lunar Ephemeris DE 421." *IPN Progress Report* 42-178, 2009.

[2] Ma, C. et al. "The International Celestial Reference Frame as Realized by Very Long Baseline Interferometry." *Astronomical Journal*, Vol. 116, 516–546, 1998.

[3] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

Extended Capabilities

C/C++ Code Generation

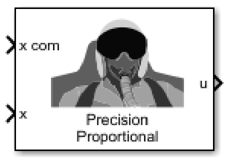
Generate C and C++ code using Simulink® Coder™.

See Also

`aeroDataPackage` | `juliandate` | Constant | Direction Cosine Matrix ECI to ECEF | Earth Nutation | Julian Date Conversion | Moon Libration

Precision Pilot Model

Represent precision pilot model



Libraries:
Aerospace Blockset / Pilot Models

Description

The Precision Pilot Model block represents the pilot model described in *Mathematical Models of Human Pilot Behavior* [1]. This pilot model is a single input, single output (SISO) model that represents some aspects of human behavior when controlling aircraft. When modeling human pilot models, use this block for more accuracy than that provided by the Tustin Pilot Model and Crossover Pilot Model blocks.

This block has non-linear behavior. If you want to linearize the block (for example, with one of the `linmod` functions), you might need to change the Pade approximation order. The Precision Pilot Model block implementation incorporates the Transport Delay block with the **Pade order (for linearization)** parameter set to `2` by default. To change this value, use the `set_param` function, for example:

```
set_param(gcb,'pade','3')
```

This block is an extension of the Crossover Pilot Model block. It implements the equation described in "Algorithms" on page 5-707.

Ports

Input

x com — Signal command
scalar

Signal command that the pilot model controls, specified as a scalar.

Data Types: `double`

x — Signal controlled by pilot
scalar

Signal that the pilot model controls, specified as a scalar.

Data Types: `double`

Output

u — Aircraft command
scalar

Aircraft command, returned as a scalar.

Data Types: `double`

Parameters

Type of control — Aircraft dynamics control

`Proportional` (default) | `Rate or velocity` | `Acceleration` | `Second order`

Aircraft dynamics control. The equalizer form changes according to these values. For more information, see [2]. To help you decide, this table lists the options and associated dynamics.

Option (Controlled Element Transfer Function)	Transfer Function of Controlled Element (Y_c)	Transfer Function of Pilot (Y_p)
Proportional	K_c	Lag-lead, $T_I >> T_L$
Rate or velocity	$\frac{K_c}{s}$	1
Acceleration	$\frac{K_c}{s^2}$	Lead-lag, $T_L >> T_I$
Second order	$\frac{K_c \omega_n{}^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$	Lead-lag if $\omega_m << 2/\tau$. Lag-lead if $\omega_m >> 2/\tau$.

This table defines the variables used in the list of control options.

Variable	Description
K_c	Aircraft gain.
T_I	Lag constant.
T_L	Lead constant.
ζ	Damping ratio for the aircraft.
ω_n	Natural frequency of the aircraft.

Programmatic Use
Block Parameter: `sw_popup`
Type: character vector
Values: `'Proportion'` | `'Rate or velocity'` | `'Acceleration'` | `'Second order'`
Default: `'Proportion'`

Pilot gain — Pilot gain

1 (default) | scalar

Pilot gain, specified as a double scalar.

Programmatic Use
Block Parameter: `Kp`
Type: character vector
Values: double scalar

Default: '1'

Pilot time delay(s) — Pilot time delay

0.1 (default) | scalar

Total pilot time delay, specified as a double scalar, in seconds. This value typically ranges from 0.1 s to 0.2 s.

Programmatic Use
Block Parameter: time_delay
Type: character vector
Values: double scalar
Default: '0.1'

Equalizer lead constant — Equalizer lead constant

1 (default) | scalar

Equalizer lead constant, specified as a double scalar.

Dependencies

To enable this parameter, set **Type of control** to Proportional, Acceleration, or Second order.

Programmatic Use
Block Parameter: TL
Type: character vector
Values: double scalar
Default: '1'

Equalizer lag constant — Equalizer lag constant

5 (default) | scalar

Equalizer lag constant, specified as a double scalar.

Dependencies

To enable this parameter, set **Type of control** to Proportional, Acceleration, or Second order.

Programmatic Use
Block Parameter: TI
Type: character vector
Values: double scalar
Default: '5'

Lag constant for neuromuscular system — Lag constant

0.1 (default) | scalar

Neuromuscular system lag constant, specified as a double scalar.

Programmatic Use
Block Parameter: `TN1`
Type: character vector
Values: double scalar
Default: `0.1`

Undamped natural frequency neuromuscular system (rad/s) — Undamped natural frequency

`20` (default) | scalar

Undamped natural frequency of the neuromuscular system, specified as a double scalar, in rad/s.

Programmatic Use
Block Parameter: `nat_freq`
Type: character vector
Values: double scalar
Default: `20`

Damping neuromuscular system — Damping neuromuscular system

`0.7` (default) | scalar

Damping neuromuscular system, specified as a double scalar.

Programmatic Use
Block Parameter: `damp`
Type: character vector
Values: double scalar
Default: `0.7`

Controlled element undamped natural frequency (rad/s) — Controlled element undamped natural frequency

`15` (default) | scalar

Controlled element undamped natural frequency, specified as a double scalar, in rad/s.

Dependencies

To enable this parameter, set **Type of control** to `Second order`.

Programmatic Use
Block Parameter: `omega_m`
Type: character vector
Values: double scalar
Default: `15`

Algorithms

When calculating the model, this block also takes into account the neuromuscular dynamics of the pilot. This block implements the following equation:

$$Y_p = K_p e^{-\tau s}\left(\frac{T_L s + 1}{T_I s + 1}\right)\left[\frac{1}{(T_{N1}s + 1)\left(\frac{s^2}{\omega_N{}^2} + \frac{2\zeta_N}{\omega_N}s + 1\right)}\right],$$

where:

Variable	Description
K_p	Pilot gain.
τ	Pilot delay time.
T_L	Time lead constant for the equalizer term.
T_I	Time lag constant.
T_{N1}	Time constant for the neuromuscular system.
ω_N	Undamped frequency for the neuromuscular system.
ζ_N	Damping ratio for the neuromuscular system.

A sample value for the natural frequency and the damping ratio of a human is 20 rad/s and 0.7, respectively. The term containing the lead-lag term is the equalizer form. This form changes depending on the characteristics of the controlled system. A consistent behavior of the model can occur at different frequency ranges other than the crossover frequency.

Version History

Introduced in R2012b

References

[1] McRuer, D. T., Krendel, E., *Mathematical Models of Human Pilot Behavior*. Advisory Group on Aerospace Research and Development AGARDograph 188, Jan. 1974.

[2] McRuer, D. T., Graham, D., Krendel, E., and Reisener, W., *Human Pilot Dynamics in Compensatory Systems*. Air Force Flight Dynamics Lab. AFFDL-65-15. 1965.

Extended Capabilities

C/C++ Code Generation

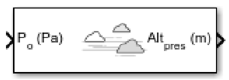
Generate C and C++ code using Simulink® Coder™.

See Also

Crossover Pilot Model | Tustin Pilot Model | Transport Delay | `linmod`

Pressure Altitude

Calculate pressure altitude based on ambient pressure



Libraries:
Aerospace Blockset / Environment / Atmosphere

Description

The Pressure Altitude block computes the pressure altitude based on ambient pressure. Pressure altitude is the altitude in the 1976 Committee on the Extension of the Standard Atmosphere (COESA) United States with specified ambient pressure.

Pressure altitude is also known as the mean sea level (MSL) altitude.

The Pressure Altitude block icon port label change based on the input and output units selected from the **Units** list.

Limitations

- Below the pressure of 0.3961 Pa (approximately 0.00006 psi) and above the pressure of 101325 Pa (approximately 14.7 psi), altitude values are extrapolated logarithmically.
- Air is assumed to be dry and an ideal gas.

Ports

Input

Port_1 — Static pressure
scalar | array

Static pressure, specified as a scalar or array.

Data Types: `double`

Output

Port_1 — Pressure altitude
scalar | array

Pressure altitude, returned as a scalar or vector.

Data Types: `double`

Parameters

Units — Input units

`Metric (MKS)` (default) | `English`

Input units, specified as:

Units	Pstatic	Alt_p
`Metric (MKS)`	Pascal	Meters
`English`	Pound force per square inch	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced before R2006a

References

[1] U.S. Standard Atmosphere, 1976, U.S. Government Printing Office, Washington, D.C.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

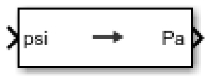
COESA Atmosphere Model

Topics

Ideal Airspeed Correction

Pressure Conversion

Convert from pressure units to desired pressure units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Pressure Conversion block computes the conversion factor from specified input pressure units to specified output pressure units and applies the conversion factor to the input signal.

The Pressure Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Pressure
scalar | array

Pressure, specified as a scalar or array, in initial pressure units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Pressure
scalar | array

Pressure, returned as a scalar or array, in final pressure units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`psi` (default) | `Pa` | `psf` | `atm`

Input units, specified as:

psi	Pound mass per square inch
Pa	Pascals
psf	Pound mass per square foot
atm	Atmospheres

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `'psi'` | `'Pa'` | `'psf'` | `'atm'`
Default: `'psi'`

Final unit — Output units

`Pa` (default) | `psi` | `psf` | `atm`

Output units, specified as:

psi	Pound mass per square inch
Pa	Pascals
psf	Pound mass per square foot
atm	Atmospheres

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `'psi'` | `'Pa'` | `'psf'` | `'atm'`
Default: `'Pa'`

Version History
Introduced before R2006a

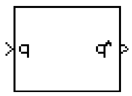
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Temperature Conversion | Velocity Conversion

Quaternion Conjugate

Calculate conjugate of quaternion



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Conjugate block calculates the conjugate for a given quaternion. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For more information on quaternion forms, see “Algorithms” on page 5-713

Ports

Input

q — Input quaternion
quaternion | vector of quaternions

Quaternions in the form of [q_0, r_0, ... , q_1, r_1, ... , q_2, r_2, ... , q_3, r_3, ...], specified as a quaternion or vector.

Data Types: `double` | `bus`

Output

q' — Quaternion conjugate
quaternion conjugate | vector of quaternion conjugates

Quaternion conjugates in the form of [q_0', r_0', ... , q_1', r_1', ... , q_2', r_2', ... , q_3', r_3', ...], returned as a quaternion or vector.

Data Types: `double` | `bus`

Algorithms

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 .$$

The quaternion conjugate has the form of

$$q' = q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3 .$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Division | Quaternion Inverse | Quaternion Modulus | Quaternion Multiplication | Quaternion Norm | Quaternion Rotation | Quaternion Normalize

Quaternion Division

Divide quaternion by another quaternion

Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Division block divides a given quaternion by another. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. The output is the resulting quaternion from the division or vector of resulting quaternions from division. For the quaternion forms used, see “Algorithms” on page 5-715.

Ports

Input

q — Dividend quaternion
quaternion | vector of quaternions

Dividend quaternions in the form of [q_0, p_0, ..., q_1, p_1, ... , q_2, p_2, ... , q_3, p_3, ...], specified as a quaternion or vector of quaternions.

Data Types: `double`

r — Divisor quaternion
quaternion | vector of quaternions

Divisor quaternions in the form of [s_0, r_0, ..., s_1, r_1, ... , s_2, r_2, ... , s_3, r_3, ...], specified as a quaternion or vector of quaternions.

Data Types: `double`

Output

q/r — Output quaternion
quaternion | vector

Output quaternion or vector of resulting quaternions from division.

Data Types: `double`

Algorithms

The quaternions have the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

and

$$r = r_0 + \mathbf{i}r_1 + \mathbf{j}r_2 + \mathbf{k}r_3\,.$$

The resulting quaternion from the division has the form of

$$t = \frac{q}{r} = t_0 + \mathbf{i}t_1 + \mathbf{j}t_2 + \mathbf{k}t_3,$$

where

$$t_0 = \frac{(r_0q_0 + r_1q_1 + r_2q_2 + r_3q_3)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$
$$t_1 = \frac{(r_0q_1 - r_1q_0 - r_2q_3 + r_3q_2)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$
$$t_2 = \frac{(r_0q_2 + r_1q_3 - r_2q_0 - r_3q_1)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$
$$t_3 = \frac{(r_0q_3 - r_1q_2 + r_2q_1 - r_3q_0)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

C/C++ Code Generation

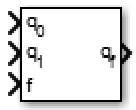
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Conjugate | Quaternion Inverse | Quaternion Modulus | Quaternion Multiplication | Quaternion Norm | Quaternion Normalize | Quaternion Rotation

Quaternion Interpolation

Quaternion interpolation between two quaternions



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Interpolation block calculates the quaternion interpolation between two normalized quaternions by an interval fraction. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. Select the interpolation method from SLERP, LERP, or NLERP. For equations used for the interpolation methods, see “Algorithms” on page 5-719.

The two normalized quaternions are the two extremes between which the block calculates the quaternion.

Ports

Input

q_0 — First normalized quaternion
4-by-1 vector | 1-by-4 vector

First normalized quaternion for which to calculate the interpolation. This quaternion must be a normalized quaternion

Data Types: `double`

q_1 — Second normalized quaternion
4-by-1 vector | 1-by-4 vector

Second normalized quaternion for which to calculate the interpolation, specified as a 4-by-1 vector or 1-by-4 vector. This quaternion must be a normalized quaternion.

Data Types: `double`

f — Interval fraction
scalar

Interval fraction by which to calculate the quaternion interpolation. This value varies between 0 and 1. It represents the intermediate rotation of the quaternion to be calculated. This fraction affects the interpolation method rotational velocities.

Dependencies

The interval fraction affects the rotational velocities of the interpolation methods for the **Methods** parameter. For more information on interval fractions, see [1].

Data Types: `double`

Output

q_f — Natural logarithm
vector

Natural logarithm of quaternion, returned as a vector.

Data Types: `double`

Parameters

Methods — Quaternion interpolation method

SLERP (default) | LERP | NLERP

Quaternion interpolation method to calculate the quaternion interpolation, specified as:

- `SLERP`

  Quaternion slerp. Spherical linear quaternion interpolation method.
- `LERP`

  Quaternion lerp. Linear quaternion interpolation method.
- `NLERP`

  Normalized quaternion linear interpolation method.

Dependencies

These methods have different rotational velocities, depending on the interval fraction from input port f. For more information on interval fractions, see [1].

Programmatic Use
Block Parameter: `method`
Type: character vector
Values: `'SLERP'` | `'LERP'` | `'NLERP'`
Default: `'SLERP'`

Action for out-of-range input — Out-of-range block behavior

`None` (default) | `Warning` | `Error`

Out-of-range block behavior, specified as:

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`

Default: `'Warning'`

Algorithms

$Slerp(p, q, h) = p(p^{*}q)^{h}$ with $h \in [0, 1]$.

$LERP(p, q, h) = p(1 - h) + qh$ with $h \in [0, 1]$.

With $r = LERP(p, q, h), NLERP(p, q, h) = \frac{r}{|r|}$.

Version History

Introduced in R2016a

References

[1] Dam, Erik B., Martin Koch, Martin Lillholm. "Quaternions, Interpolation, and Animation." University of Copenhagen, København, Denmark, 1998.

Extended Capabilities

C/C++ Code Generation

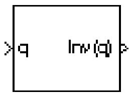
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Normalize

Quaternion Inverse

Calculate inverse of quaternion



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Inverse block calculates the inverse for a given quaternion. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For the equations used for the quaternion and quaternion inverse, “Algorithms” on page 5-720.

Ports

Input

q — Quaternion
quaternion | vector of quaternions

Quaternions in the form of [q_0 , r_0 , ..., q_1 , r_1 , ... , q_2 , r_2 , ... , q_3 , r_3 , ...], specified as a quaternion or vector of quaternions.

Data Types: `double`

Output

Inv(q) — Quaternion inverse
quaternion inverse | vector of quaternion inverses

Quaternion inverse or vector of quaternion inverses.

Data Types: `double`

Algorithms

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 .$$

The quaternion inverse has the form of

$$q^{-1} = \frac{q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3}{q_0^2 + q_1^2 + q_2^2 + q_3^2} .$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

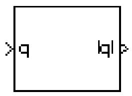
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Rotation | Quaternion Normalize | Quaternion Norm | Quaternion Multiplication | Quaternion Modulus | Quaternion Division | Quaternion Conjugate

Quaternion Modulus

Calculate modulus of quaternion



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Modulus block calculates the magnitude for a given quaternion. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For the equations used for the quaternion and quaternion modulus, see “Algorithms” on page 5-722.

Ports

Input

q — Quaternion
quaternion | vector of quaternions

Quaternions in the form of [q_0 , r_0 , ..., q_1 , r_1 , ... , q_2 , r_2 , ... , q_3 , r_3 , ...], specified as a quaternion or vector of quaternions.

Data Types: `double`

Output

|q| — Quaternion modulus
quaternion modulus | vector of quaternion moduli

Quaternion modulus or vector of quaternion moduli in the form of [| q |, | r |, ...].

Data Types: `double`

Algorithms

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 .$$

The quaternion modulus has the form of

$$|q| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

C/C++ Code Generation

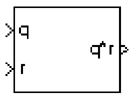
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Conjugate | Quaternion Rotation | Quaternion Normalize | Quaternion Norm | Quaternion Multiplication | Quaternion Inverse | Quaternion Division

Quaternion Multiplication

Calculate product of two quaternions



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Multiplication block calculates the product for two given quaternions. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For more information on the quaternion forms, see “Algorithms” on page 5-724.

Ports

Input

q — First quaternion
quaternion | vector of quaternions

First quaternion, specified as a vector or vector of quaternions. A vector of quaternions has this form, where *q* and *p* are quaternions:

$$[q_0, p_0, ..., q_1, p_1, ..., q_2, p_2, ..., q_3, p_3, ...]$$

Data Types: `double`

r — Second quaternion
quaternion | vector of quaternions

Second quaternion, specified as a vector or vector of quaternions. A vector of quaternions has this form, where *s* and *r* are quaternions:

$$[s_0, r_0, ..., s_1, r_1, ..., s_2, r_2, ..., s_3, r_3, ...]$$

Data Types: `double`

Output

q*r — Product
vector | vector of quaternion products

Product of two quaternions, output as a vector or vector of quaternion products.

Data Types: `double`

Algorithms

This block uses quaternions of the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

and

$$r = r_0 + \mathbf{i}r_1 + \mathbf{j}r_2 + \mathbf{k}r_3\,.$$

The quaternion product has the form of

$$t = q \times r = t_0 + \mathbf{i}t_1 + \mathbf{j}t_2 + \mathbf{k}t_3,$$

where

$$\begin{aligned}
t_0 &= (r_0q_0 - r_1q_1 - r_2q_2 - r_3q_3)\\
t_1 &= (r_0q_1 + r_1q_0 - r_2q_3 + r_3q_2)\\
t_2 &= (r_0q_2 + r_1q_3 + r_2q_0 - r_3q_1)\\
t_3 &= (r_0q_3 - r_1q_2 + r_2q_1 + r_3q_0)
\end{aligned}$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

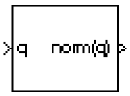
Quaternion Conjugate | Quaternion Division | Quaternion Inverse | Quaternion Modulus | Quaternion Norm | Quaternion Normalize | Quaternion Rotation

Topics

“Explore the NASA HL-20 Model” on page 1-5

Quaternion Norm

Calculate norm of quaternion



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Norm block calculates the norm for a given quaternion. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For the equations used for the quaternion and quaternion norm, see "Algorithms" on page 5-726.

Ports

Input

q — Quaternion norm
quaternion norm | vector of quaternion norms

Quaternions in the form of [q_0 , r_0 , ..., q_1 , r_1 , ... , q_2 , r_2 , ... , q_3 , r_3 , ...], specified as a quaternion norm or vector of quaternion norms.

Data Types: `double`

Output

norm(q) — Quaternion norm
quaternion norm | vector of quaternion norms

Quaternion norm or vector of quaternion norms in the form of [*norm* (q), *norm* (r), ...].

Data Types: `double`

Algorithms

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 .$$

The quaternion norm has the form of

$$norm(q) = q_0^2 + q_1^2 + q_2^2 + q_3^2$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

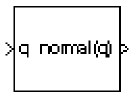
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Conjugate | Quaternion Division | Quaternion Inverse | Quaternion Modulus | Quaternion Multiplication | Quaternion Normalize | Quaternion Rotation

Quaternion Normalize

Normalize quaternion



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Normalize block calculates a normalized quaternion for a given quaternion. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For the equations used for the quaternion and normalized quaternion, see “Algorithms” on page 5-728.

Ports

Input

q — Quaternion
quaternion | vector of quaternions

Quaternions in the form of [q_0 , r_0 , ..., q_1 , r_1 , ... , q_2 , r_2 , ... , q_3 , r_3 , ...], specified as a quaternion or vector of quaternions.

Data Types: `double`

Output

normal(q) — Normalized quaternion
normalized quaternion | vector of normalized quaternions

Normalized quaternion or vector of normalized quaternions.

Data Types: `double`

Algorithms

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 .$$

The normalized quaternion has the form of

$$normal(q) = \frac{q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3}{\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}} .$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

Extended Capabilities

C/C++ Code Generation

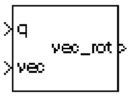
Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Conjugate | Quaternion Division | Quaternion Inverse | Quaternion Modulus | Quaternion Multiplication | Quaternion Norm | Quaternion Rotation

Quaternion Rotation

Rotate vector by quaternion



Libraries:
Aerospace Blockset / Utilities / Math Operations

Description

The Quaternion Rotation block calculates the resulting vector following the passive rotation of initial vector `vec` by quaternion `q` and returns a final vector, the rotated vector or vector of rotated vectors. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. This block normalizes all quaternion inputs. For the equations used for the quaternion, initial vector, and final vector, see “Algorithms” on page 5-730.

Ports

Input

q — Quaternion
quaternion | vector

Quaternions in the form of [q_0, r_0, ..., q_1, r_1, ... , q_2, r_2, ... , q_3, r_3, ...], specified as a quaternion or vector of quaternions.

Data Types: `double`

vec — Initial vector
vector | vector of vectors

Initial vector or vector of vectors in the form of [v_1, u_1, ... , v_2, u_2, ... , v_3, u_3, ...].

Data Types: `double`

Output

vec_rot — Final quaternion
rotated quaternion | vector of rotated quaternions

Final vector or vector of rotated vectors.

Data Types: `double`

Algorithms

The normalized quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 .$$

The vector has the form of

$$v = \mathbf{i}v_1 + \mathbf{j}v_2 + \mathbf{k}v_3\,.$$

The Aerospace Blockset defines a passive quaternion rotation of the form:

$$v' = q^{-1} \otimes \begin{bmatrix} 0 \\ v \end{bmatrix} \otimes q,$$

where ⊗ is the operator of a quaternion multiplication.

The final vector has the form of

$$v' = \begin{bmatrix} v_1{'} \\ v_2{'} \\ v_3{'} \end{bmatrix} = \begin{bmatrix} (1-2q_2^2-2q_3^2) & 2(q_1q_2+q_0q_3) & 2(q_1q_3-q_0q_2) \\ 2(q_1q_2-q_0q_3) & (1-2q_1^2-2q_3^2) & 2(q_2q_3+q_0q_1) \\ 2(q_1q_3+q_0q_2) & 2(q_2q_3-q_0q_1) & (1-2q_1^2-2q_2^2) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, Second Edition. Hoboken, NJ: Wiley–Interscience.

[2] Diebel, James. "Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors." Stanford University, Stanford, California, 2006.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Quaternion Conjugate | Quaternion Division | Quaternion Inverse | Quaternion Multiplication | Quaternion Norm | Quaternion Normalize

Topics

“Passive Transformations” on page 2-9

Quaternions to Direction Cosine Matrix

Convert quaternion vector to direction cosine matrix

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Quaternions to Direction Cosine Matrix block transforms a four-element unit quaternion vector (q_0, q_1, q_2, q_3) into a 3-by-3 direction cosine matrix (DCM). The outputted DCM performs the coordinate transformation of a vector in inertial axes to a vector in body axes. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. This block normalizes all quaternion inputs. The quaternion input and the resulting direction cosine matrix represent a right-hand passive transformation from frame A to frame B. For more information, see “Algorithms” on page 5-732.

Ports

Input

q — Quaternion
4-by-1 vector

Quaternion, specified as a 4-by-1 vector.

Data Types: `double`

Output

DCM$_{be}$ — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, returned as a 3-by-3 matrix.

Data Types: `double`

Algorithms

Using quaternion algebra, if a point P is subject to the rotation described by a quaternion q, it changes to P' given by the following relationship:

$$P' = qPq^c$$
$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$
$$q^c = q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3$$
$$P = 0 + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z$$

Expanding P' and collecting terms in x, y, and z gives the following for P' in terms of P in the vector quaternion format:

$$P' = \begin{bmatrix} 0 \\ x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 0 \\ (q_0^2 + q_1^2 - q_2^2 - q_3^2)x + 2(q_1q_2 - q_0q_3)y + 2(q_1q_3 + q_0q_2)z \\ 2(q_0q_3 + q_1q_2)x + (q_0^2 - q_1^2 + q_2^2 - q_3^2)y + 2(q_2q_3 - q_0q_1)z \\ 2(q_1q_3 - q_0q_2)x + 2(q_0q_1 + q_2q_3)y + (q_0^2 - q_1^2 - q_2^2 + q_3^2)z \end{bmatrix}$$

Since individual terms in P' are linear combinations of terms in x, y, and z, a matrix relationship to rotate the vector (x, y, z) to (x', y', z') can be extracted from the preceding. This matrix rotates a vector in inertial axes, and hence is transposed to generate the DCM that performs the coordinate transformation of a vector in inertial axes into body axes.

$$DCM = \begin{bmatrix} (q_0^2 + q_1^2 - q_2^2 - q_3^2) & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & (q_0^2 - q_1^2 + q_2^2 - q_3^2) & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & (q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix}$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Quaternions | Rotation Angles to Direction Cosine Matrix | Rotation Angles to Quaternions

Quaternions to Rodrigues

Convert quaternion to Euler-Rodrigues vector

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Quaternions to Rodrigues block converts the 4-by-1 quaternion to the three-element Euler-Rodrigues (Rodrigues) vector. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. This block normalizes all quaternion inputs. The quaternion input and the resulting Euler-Rodrigues vector represent a right-hand passive transformation from frame A to frame B. For more information on Euler-Rodrigues vectors, see “Algorithms” on page 5-734.

Ports

Input

Quaternion — Quaternion
4-by-1 matrix

Quaternion from which to determine Euler-Rodrigues vector. Quaternion scalar is the first element.

Data Types: `double`

Output

rod — Euler-Rodrigues vector
three-element vector

Euler-Rodrigues vector determined from the quaternion.

Data Types: `double`

Algorithms

- An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

  $$\vec{b} = [b_x \; b_y \; b_z]$$

  where:

  $$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

  $$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

  $$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

Version History

Introduced in R2017a

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

Extended Capabilities

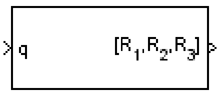
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rodrigues | Rodrigues to Direction Cosine Matrix | Rodrigues to Quaternions | Rodrigues to Rotation Angles | Rotation Angles to Rodrigues

Quaternions to Rotation Angles

Determine rotation vector from quaternion



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Quaternions to Rotation Angles block converts the four-element quaternion vector (q_0, q_1, q_2, q_3), into the rotation described by the three rotation angles (R1, R2, R3). The block generates the conversion by computing elements in the direction cosine matrix (DCM) as a function of the rotation angles. The elements in the DCM are functions of a unit quaternion vector. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. This block normalizes all quaternion inputs. The resulting rotation angles represent a series of right-hand intrinsic passive rotations from frame A to frame B. The quaternion represents a right-hand passive transformation from frame A to frame B. For more information on the direction cosine matrix, see “Algorithms” on page 5-737.

Limitations

- For the ZYX, ZXY, YXZ, YZX, XYZ, and XZY rotations, the block generates an R2 angle that lies between ±pi/2 radians, and R1 and R3 angles that lie between ±pi radians.
- For the 'ZYZ', 'ZXZ', 'YXY', 'YZY', 'XYX', and 'XZX' rotations, the block generates an R2 angle that lies between 0 and pi radians, and R1 and R3 angles that lie between ±pi radians. However, in the latter case, when R2 is 0, R3 is set to 0 radians.

Ports

Input

q — Quaternion
4-by-1 vector

Quaternion, specified as a 4-by-1 vector.

Data Types: `double`

Output

[R_1,R_2,R_3] — Rotation angles
3-by-1 vector

Rotation angles, returned 3-by-1 vector, in radians.

Data Types: `double`

Parameters

Rotation order — Rotation order

ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Output rotation order for the three rotation angles.

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: ZYX | ZYZ |ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX
Default: `'ZYX'`

Algorithms

The elements in the DCM are functions of a unit quaternion vector. For example, for the rotation order *z-y-x*, the DCM is defined as:

$$DCM = \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ (\sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi) & (\sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi) & \sin\phi\cos\theta \\ (\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi) & (\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi) & \cos\phi\cos\theta \end{bmatrix}$$

The DCM defined by a unit quaternion vector is:

$$DCM = \begin{bmatrix} (q_0^2 + q_1^2 - q_2^2 - q_3^2) & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & (q_0^2 - q_1^2 + q_2^2 - q_3^2) & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & (q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix}$$

From the preceding equation, you can derive the following relationships between DCM elements and individual rotation angles for a ZYX rotation order:

$$\begin{aligned} \phi &= \operatorname{atan}(DCM(2,3), DCM(3,3)) \\ &= \operatorname{atan}(2(q_2q_3 + q_0q_1), (q_0^2 - q_1^2 - q_2^2 + q_3^2)) \\ \theta &= \operatorname{asin}(-DCM(1,3)) \\ &= \operatorname{asin}(-2(q_1q_3 - q_0q_2)) \\ \psi &= \operatorname{atan}(DCM(1,2), DCM(1,1)) \\ &= \operatorname{atan}(2(q_1q_2 + q_0q_3), (q_0^2 + q_1^2 - q_2^2 - q_3^2)) \end{aligned}$$

where Ψ is R1, Θ is R2, and Φ is R3.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

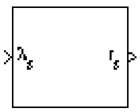
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Quaternions | Quaternions to Direction Cosine Matrix | Rotation Angles to Direction Cosine Matrix | Rotation Angles to Quaternions

Radius at Geocentric Latitude

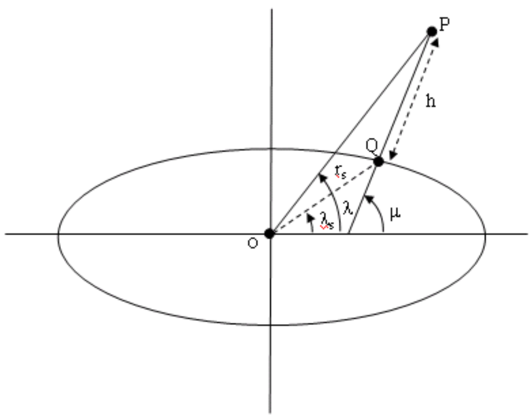
Estimate radius of ellipsoid planet at geocentric latitude



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Radius at Geocentric Latitude block estimates the radius (r_s) of an ellipsoid planet at a particular geocentric latitude (λ_s).



The following equation estimates the ellipsoid radius (r_s) using flattening ($\bar{f}$), geocentric latitude ($\bar{\lambda}_s$), and equatorial radius ($\bar{R}$):

$$r_s = \sqrt{\frac{R^2}{1 + \left[1/(1 - f)^2 - 1\right]\sin^2\lambda_s}}.$$

Ports

Input

λ_s — Geocentric latitude
scalar | vector

Geocentric latitude, specified as a scalar or vector, in degrees.

Data Types: `double`

Output

r_s — Radius
scalar | vector

Radius of planet at geocentric latitude, returned as a scalar or vector, in the same units as flattening.

Data Types: `double`

Parameters

Units — Output units

`Metric (MKS)` (default) | `English`

Output units, specified as:

Units	Equatorial Radius	Radius at Geocentric Latitude
`Metric (MKS)`	Meters	Meters
`English`	Feet	Feet

Dependencies

To enable this parameter, set **Planet model** to `Earth (WGS84)`.

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `F`
Type: character vector
Values: double scalar
Default: `1/298.257223563`

Equatorial radius of planet — Radius of planet at equator

6378137 (default) | scalar

Radius of the planet at its equator, in the same units as the desired units for ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `6378137`

Version History

Introduced before R2006a

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, New York, John Wiley & Sons, 1992.

[2] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. Second Edition. Reston, VA: AIAA Education Series, 2000.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

ECEF Position to LLA | Direction Cosine Matrix ECEF to NED | Direction Cosine Matrix ECEF to NED to Latitude and Longitude | Geocentric to Geodetic Latitude | Geodetic to Geocentric Latitude | LLA to ECEF Position

Receive net_ctrl Packet from FlightGear

Receive `net_ctrl` packet from FlightGear

Libraries:
Aerospace Blockset / Animation / Flight Simulator Interfaces

Description

The Receive net_ctrl Packet from FlightGear block receives a network control and environment data packet, `net_ctrl`, from the simulation of a Simulink model in the FlightGear simulator, or from a FlightGear session. This data packet is compatible with a particular version of FlightGear flight simulator. This block supports all signals supported by the FlightGear `net_ctrl` data packet. The block arranges the signals into multiple groups. The block inserts zeros for packet values that are part of inactive signal groups.

The Aerospace Blockset product supports FlightGear versions starting from v2.6. If you are using a FlightGear version older than 2.6, the model displays a notification from the Simulink Upgrade Advisor. Consider using the Upgrade Advisor to upgrade your FlightGear version. For more information, see “Supported FlightGear Versions” on page 2-20.

If you run a model that contains this block in Rapid Accelerator mode, the block produces zeros (`0`s) and it does not produce deployable code. In Accelerator mode, the block works as expected.

For details on signals and signal groups, see “Output” on page 5-742.

Ports

Output

net_ctrl — Controls information from FlightGear
744-by-1 vector

Controls information from FlightGear, returned as a 744-by-1 vector.

Data Types: `uint8`

Rx bytes — Received FlightGear packet size
`0` | `744`

Received FlightGear packet size, specified as a scalar.

- 0, if no data is received
- Size of the packet (744) in bytes.

Dependencies

This port is enabled by the **Enable received flag port** check box.

Data Types: `double`

Parameters

Origin IP address — Origin IP address

`127.0.0.1` (default) | scalar

Enter a valid IP address as a dot-decimal string. This IP address must be the address of the computer from which FlightGear is run, for example, `10.10.10.3`.

You can also use a MATLAB expression that returns a valid IP address as a character vector. If FlightGear is run on the local computer, leave the default value of `127.0.0.1` (localhost).

To determine the source IP address, you can use one of several techniques, such as:

- Use 127.0.0.1 for the local computer (localhost).
- Ping another computer from a Windows `cmd.exe` (or Linux shell) prompt:

```
C:\> ping andyspc

Pinging andyspc [144.213.175.92] with 32 bytes of data:

Reply from 144.213.175.92: bytes=32 time=30ms TTL=253
Reply from 144.213.175.92: bytes=32 time=20ms TTL=253
Reply from 144.213.175.92: bytes=32 time=20ms TTL=253
Reply from 144.213.175.92: bytes=32 time=20ms TTL=253

Ping statistics for 144.213.175.92:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 20ms, Maximum =  30ms, Average =  22ms
```

- On a Windows machine, type `ipconfig` and use the returned IP address:

```
H:\>ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . :
        IP Address. . . . . . . . . . . . : 192.168.42.178
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 192.168.42.254
```

Programmatic Use
Block Parameter: `ReceiveAddress`
Type: character vector
Values: scalar
Default: `'127.0.0.1'`

Origin port — Origin port

5505 (default)

UDP port that the block accepts data from. The sender sends data to the port specified in this parameter. This value must match the **Origin port** parameter of the Generate Run Script block. It must be a unique port number that no other application on the computer uses. The site, `https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`, lists commonly known UDP port numbers. To identify UDP port numbers already in use on your computer, type:

```
netstat -a -p UDP
```

Programmatic Use
Block Parameter: `ReceivePort`
Type: character vector
Values: scalar
Default: `'5505'`

Sample time — Sample time

1/30 (default) | scalar

Specify the sample time (-1 for inherited).

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'1/30'`

Enable received flag port — Enable received flag output port

`off` (default) | `on`

Enable a received flag output port. Use this check box to determine if a FlightGear network packet has been received.

Dependencies

Selecting this check box enables the **Rx bytes** port.

Programmatic Use
Block Parameter: `packetFlag`
Type: character vector
Values:`'off' | 'on'`
Default: `'off'`

Version History

Introduced in R2012a

See Also

FlightGear Preconfigured 6DoF Animation | Generate Run Script | Pack net_fdm Packet for FlightGear | Send net_fdm Packet to FlightGear | Unpack net_ctrl Packet from FlightGear

Topics

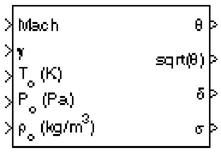
“Flight Simulator Interface” on page 2-20
“Work with the Flight Simulator Interface” on page 2-24

External Websites

https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Relative Ratio

Calculate relative atmospheric ratios



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Relative Ratio block computes the relative atmospheric ratios, including the relative temperature ratio (θ), $\sqrt{\theta}$, relative pressure ratio (δ), and relative density ratio (σ).

θ represents the ratio of the air stream temperature at a chosen reference station relative to sea level standard atmospheric conditions:

$$\theta = \frac{T}{T_0}.$$

δ represents the ratio of the air stream pressure at a chosen reference station relative to sea level standard atmospheric conditions:

$$\delta = \frac{P}{P_0}.$$

σ represents the ratio of the air stream density at a chosen reference station relative to sea level standard atmospheric conditions:

$$\sigma = \frac{\rho}{\rho_0}.$$

The Relative Ratio block icon displays the input units selected from the **Units** parameter.

Limitations

For cases in which total temperature, total pressure, or total density ratio is desired (Mach number is nonzero), the total temperature, total pressure, and total densities are calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

Ports

Input

Mach — Mach number
scalar

Mach number, specified as a scalar.

Data Types: `double`

γ — Ratio
scalar

Ratio between the specific heat at constant pressure (C_p) and the specific heat at constant volume (C_v), specified as a scalar. For example, ($\gamma = C_p/C_v$).

Data Types: `double`

T_o — Static temperature
scalar

Static temperature, specified as a scalar.

Data Types: `double`

P_o — Static pressure
scalar

Static pressure, specified as a scalar.

Data Types: `double`

ρ_o — Static density
scalar

Static density, specified as a scalar.

Data Types: `double`

Output

θ — Relative temperature ratio
scalar

Relative temperature ratio (θ), returned as a scalar.

Dependencies

To enable this port, select **Theta**.

Data Types: `double`

sqrt(θ) — Square root of relative temperature ratio
scalar

Square root of the relative temperature ratio ($\sqrt{\theta}$), returned as a scalar.

Dependencies

To enable this port, select **Square root of theta**.

Data Types: `double`

δ — Relative pressure ratio
scalar

Relative pressure ratio, (δ), returned as a scalar.

Dependencies

To enable this port, select **Delta**.

Data Types: `double`

σ — Relative density ratio
scalar

Relative density ratio, (σ), returned as a scalar.

Dependencies

To enable this port, select **Sigma**.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Input units, specified as:

Units	Tstatic	Pstatic	rho_static
`Metric (MKS)`	Kelvin	Pascal	Kilograms per cubic meter
`English`	Degrees Rankine	Pound force per square inch	Slug per cubic foot

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Theta — Relative temperature ratio

`on` (default) | `off`

When selected, the block calculates the relative temperature ratio (θ) and static temperature is a required input.

Programmatic Use
Block Parameter: `theta`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Square root of theta — Square root of relative temperature ratio

`on` (default) | `off`

When selected, the block calculates the square root of relative temperature ratio ($\sqrt{\theta}$) and static temperature is a required input.

Dependencies

Selecting this check box enables the sqrt(θ) output port.

Programmatic Use
Block Parameter: `sq_theta`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Delta — Relative pressure ratio

on (default) | `off`

When selected, the block calculates the relative pressure ratio (δ) and static pressure is a required input.

Programmatic Use
Block Parameter: `delta`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Sigma — Relative density ratio

on (default) | `off`

When selected, the block the relative density ratio (σ) and static density is a required input.

Programmatic Use
Block Parameter: `sigma`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Version History

Introduced before R2006a

References

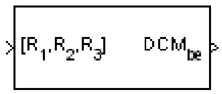
[1] *Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Rotation Angles to Direction Cosine Matrix

Convert rotation angles to direction cosine matrix



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Rotation Angles to Direction Cosine Matrix block determines the direction cosine matrix (DCM) from a given set of rotation angles, R1, R2, and R3. For example, the default rotation angle order ZYX represents a sequence where R1 is *z*-axis rotation (yaw), R2 is *y*-axis rotation (pitch), and R3 is *x*-axis rotation (roll). Use the **Rotation Order** parameter to change the sequence. The rotation angles represent a series of right-hand intrinsic passive transformation from frame A to frame B. The resulting direction cosine matrix represents a right-hand rotation from frame A to frame B.

Ports

Input

[R_1,R_2,R_3] — Rotation angles
3-by-1 vector

Rotation angles, specified as a 3-by-1 vector, in radians.

Data Types: `double`

Output

DCM_{be} — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix that performs coordinate transformations based on rotation angles, returned as a 3-by-3 matrix.

Data Types: `double`

Parameters

Rotation Order — Rotation order

ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Input rotation order for the three rotation angles.

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: `'ZYX'` | `'ZYZ'` |`'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Default: `'ZYX'`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Quaternions | Direction Cosine Matrix to Rotation Angles | Quaternions to Direction Cosine Matrix

Rotation Angles to Rodrigues

Convert rotation angles to Euler-Rodrigues vector

Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Rotation Angles to Rodrigues block converts the rotation described by the three rotation angles `R1,R2,R3` into the three-element Euler-Rodrigues vector. The rotation used in this block is a passive transformation between two coordinate systems. The rotation angles represent a series of right-hand intrinsic passive rotations from frame A to frame B. The resulting Euler-Rodrigues vector represents a right-hand passive rotation from frame A to frame B. For more information on Euler-Rodrigues vectors, see “Algorithms” on page 5-752.

Ports

Input

R1,R2,R3 — Rotation angles
three-element vector

Rotation angles, in radians, from which to determine the Euler-Rodrigues vector. Values must be double.

Output

rod — Euler-Rodrigues vector
three-element vector

Euler-Rodrigues vector determined from rotation angles.

Data Types: `double`

Parameters

Rotation order — Rotation order
ZYX (default) | ZYX | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation order for three wind rotation angles.

The default limitations for the `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` sequences generate an R2 angle that lies between ±pi/2 radians (± 90 degrees), and R1 and R3 angles that lie between ±pi radians (±180 degrees).

The default limitations for the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` sequences generate an R2 angle that lies between 0 and pi radians (180 degrees), and R1 and R3 angles that lie between ±pi (±180 degrees).

Rodrigues transformation is not defined for rotation angles equal to ±pi radians (±180 deg).

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: `'ZYX'` | `'ZYZ'` |`'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`
Default: `'ZYX'`

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x\ b_y\ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

Version History

Introduced in R2017a

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

Extended Capabilities

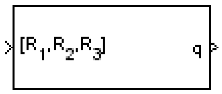
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Rodrigues | Rodrigues to Direction Cosine Matrix | Rodrigues to Quaternions | Rodrigues to Rotation Angles | Quaternions to Rodrigues

Rotation Angles to Quaternions

Calculate quaternion from rotation angles



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Rotation Angles to Quaternions block converts the rotation described by the three rotation angles (R1, R2, R3) into the four-element quaternion vector (q_0, q_1, q_2, q_3), where quaternion is defined using the scalar-first convention. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. The rotation angles represent a series of right-hand intrinsic passive transformation from frame A to frame B. The resulting quaternion represents a right-hand passive rotation from frame A to frame B. For more information on quaternions, see “Algorithms” on page 5-754.

Limitations

- The limitations for the ZYX, ZXY, YXZ, YZX, XYZ, and XZY implementations generate an R2 angle that is between ±90 degrees, and R1 and R3 angles that are between ±180 degrees.
- The limitations for the ZYZ, ZXZ, YXY, YZY, XYX, and XZX implementations generate an R2 angle that is between 0 and 180 degrees, and R1 and R3 angles that are between ±180 degrees.

Ports

Input

[R_1,R_2,R_3] — Rotation angles
3-by-1 vector

Rotation angles, specified as a 3-by-1 vector, in radians.

Data Types: `double`

Output

q — Quaternion
4-by-1 vector

Quaternion, specified as a 4-by-1 vector.

Data Types: `double`

Parameters

Rotation order — Rotation order

ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Specifies the output rotation order for three wind rotation angles.

Programmatic Use
Block Parameter: `rotationOrder`
Type: character vector
Values: `'ZYX'` | `'ZYZ'` |`'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`
Default: `'ZYX'`

Algorithms

A quaternion vector represents a rotation about a unit vector $\left(\mu_x, \mu_y, \mu_z\right)$ through the angle θ. A unit quaternion itself has unit magnitude, and can be written in the following vector format:

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2)\mu_x \\ \sin(\theta/2)\mu_y \\ \sin(\theta/2)\mu_z \end{bmatrix}$$

An alternative representation of a quaternion is as a complex number,

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

where, for the purposes of multiplication:

$$\begin{aligned} i^2 = j^2 = k^2 &= -1 \\ ij = -ji &= k \\ jk = -kj &= i \\ ki = -ik &= j \end{aligned}$$

The benefit of representing the quaternion in this way is the ease with which the quaternion product can represent the resulting transformation after two or more rotations.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

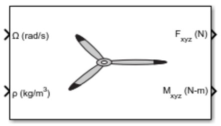
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix to Quaternions | Quaternions to Direction Cosine Matrix | Quaternions to Rotation Angles | Rotation Angles to Direction Cosine Matrix

Rotor

Rotor dynamics



Libraries:
Aerospace Blockset / Propulsion

Description

The Rotor block computes the aerodynamic forces and moments generated by a rotating propeller or rotor in all three dimensions. The block lets you include the effect of the tilt in rotor disc due to flap motion, while in forward flight, on the generated force and moments.

Limitations

- The block follows a simplified approach with the option to include steady state flap effects. It does not model dynamic flap, lag, or feathering motion of blade.
- The block does not directly consider the variation in pitch angle input, although you can perform approximate analysis by varying the blade twist.
- The block uses **Twist distribution** to model only linear or ideal twist distributions. The block assumes that the blade chord and lift curve slope are constant.

Ports

Input

Ω (rad/sec) — Rotor speed
scalar

Rotor speed, specified as a scalar in rad/sec in body frame.

Data Types: `double`

ρ — Air density
positive scalar

Air density, specified as a positive scalar in specified units.

Data Types: `double`

V_b (m/s) — Body velocity of rotor
3-by-1 vector | 1-by-3 vector

Velocity of rotor, specified as a 3-by-1 or 1-by-3 vector in m/s, in body frame. To perform a multisystem analysis, consider connecting output from a State-Space or Integrator block to this port.

Data Types: `double`

ω_b — Angular velocity
3-by-1 vector | 1-by-3 vector

Angular velocity of entire vehicle, specified as a 3-by-1 or 1-by-3 vector in rad/s in body frame.

Data Types: `double`

Output

F_{xyz}(N) — Total force
three-element vector

Total force, returned as a three-element vector in body frame. Units depend on the **Units** parameter.

Data Types: `double`

M_{xyz}(Nm) — Net moment
three-element vector

Net moment in the *x*-*y*-*z* direction, returned as a three-element vector in body frame. Units depend on the **Units** parameter.

C_T — Computed thrust coefficient
scalar positive

Computed thrust coefficient, returned as a scalar positive.

Dependencies

To enable this output port, select these check boxes:

- **Compute CT and CQ**
- **Output computed CT and CQ**

C_Q — Computed torque coefficient
scalar positive

Computed torque coefficient, returned as a scalar positive.

Dependencies

To enable this output port, select these check boxes:

- **Compute CT and CQ**
- **Output computed CT and CQ**

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Units	Density	Force	Mass	Velocity	Radius	Chord	Hinge offset
Metric (MKS)	kg/m^3	Newtons	Newton-meter	Meters per second	Meters	Meters	Meters
English (Velocity in ft/s)	$slug/ft^3$	Pound force	Pound force-feet	Feet per second	Feet	Feet	Feet
English (Velocity in kts)	$slug/ft^3$	Pound force	Pound force-feet	Feet per second	Feet	Feet	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Modeling — Rotor thrust calculation method

`Without flap effects` (default) | `With flap effects`

Rotor thrust calculation method, specified as:

- `Without flap effects` — Model rotor thrust using force and moment calculations. For more information, see "Force and Moment" on page 5-761.
- `With flap effects` — Effect of tilt in rotor disc due to flap motion, while in forward flight, is included. The steady state lateral and longitudinal flap angles are calculated using the equations from [1] and [2].

Programmatic Use
Block Parameter: `modelMode`
Type: character vector
Values: `'Without flap effects'` | `'With flap effects'`
Default: `'Without flap effects'`

Number of blades — Number of blades per rotor

2 (default) | nonzero positive scalar

Number of blades, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `Nb`
Type: character vector
Values: `'2'` | nonzero positive scalar

Default: `'2'`

Compute CT and CQ — Option to calculate thrust coefficient and torque coefficient

on (default) | off

Select this check box to let the block calculate the thrust coefficient and torque coefficient. For more information on these calculations, see “Thrust Coefficient and Torque Coefficient Calculations” on page 5-762. The block assumes the aerodynamic and structural parameters to be constant.

Clear this check box to specify thrust coefficient and torque coefficient values.

Programmatic Use
Block Parameter: `CTcheck`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Output computed CT and CQ — Output computed thrust coefficient and torque coefficient

on (default) | off

Select this check box to output the calculated thrust coefficient and torque coefficient to $\mathbf{C_T}$ and $\mathbf{C_Q}$ output ports. For more information on these calculations, see “Thrust Coefficient and Torque Coefficient Calculations” on page 5-762. The block assumes the aerodynamic and structural parameters to be constant.

Otherwise, clear this check box.

Programmatic Use
Block Parameter: `CTout`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Thrust coefficient (CT) — Thrust coefficient

`0.0107` (default) | nonzero positive scalar

Thrust coefficient, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, clear the **Compute CT and CQ** check box.

Programmatic Use
Block Parameter: `CT`
Type: character vector
Values: `'0.0107'` | nonzero positive scalar
Default: `'0.0107'`

Torque coefficient (CQ) — Torque coefficient

`7.8263e-4` (default) | nonzero positive scalar

Torque coefficient, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, clear the **Compute CT and CQ** check box.

Programmatic Use
Block Parameter: `CQ`
Type: character vector
Values: `'7.8263e-4'` | nonzero positive scalar
Default: `'7.8263e-4'`

Radius (m) — Rotor radius

`0.0330` (default) | nonzero positive scalar

Rotor radius, specified as a nonzero positive scalar.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: `'0.0330'` | nonzero positive scalar
Default: `'0.0330'`

Chord (m) — Blade chord

`0.0080` (default) | nonzero positive scalar

Blade chord, specified as a nonzero positive scalar.

Programmatic Use
Block Parameter: `c`
Type: character vector
Values: `'0.0080'` | nonzero positive scalar
Default: `'0.0080'`

Hinge offset (m) — Hinge offset

`0` (default) | positive scalar

Hinge offset, specified as a positive scalar. This value is typically `0` for propellers.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: `'0'` | positive scalar
Default: `'0'`

Lift curve slope (per rad) — Lift curve slope

`5.5` (default) | nonzero positive scalar

Lift curve slope, specified as a nonzero positive scalar. The block assumes the aerodynamic and structural parameters to be constant. The block does not consider variation with respect to angle of attack.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use

Block Parameter: `clalpha`
Type: character vector
Values: `'5.5'` | nonzero positive scalar
Default: `'5.5'`

Lock number — Lock number

`0.6051` (default) | nonzero positive scalar

Lock number, which is the ratio of aerodynamics forces to inertial forces, specified as a nonzero positive scalar.

Dependencies

To enable this parameter, set **Modeling** to `With flap effects`.

Programmatic Use

Block Parameter: `gamma`
Type: character vector
Values: `'0.6051'` | nonzero positive scalar
Default: `'0.6051'`

Twist distribution — Rotor blade twist distribution

`Linear` (default) | `Ideal`

Rotor blade twist distribution, specified as:

- `Linear` — Close approximation of blade twist distribution.
- `Ideal` — Optimal approximation of blade twist distribution.

Dependencies

To enable this parameter, select the **Compute CT and CQ** check box and set **Modeling** to `With flap effects`.

Programmatic Use

Block Parameter: `twistType`
Type: character vector
Values: `'Linear'` | `'Ideal'`
Default: `'Linear'`

Blade root angle (rad) — Blade root pitch angle

`0.2548` (default) | nonzero positive scalar

Blade root pitch angle θ_0, specified as a nonzero positive scalar.

Dependencies

To enable this parameter:

- Set **Modeling** to `With flap effects`.
- Set **Twist distribution** to `Linear`.

Programmatic Use

Block Parameter: `theta0`
Type: character vector
Values: `'0.2548'` | nonzero positive scalar
Default: `'0.2548'`

Blade twist angle (rad) — Blade twist angle

`-0.1361` (default) | positive scalar

Blade twist angle θ_1, specified as a positive scalar.

Dependencies

To enable this parameter:

- Set **Modeling** to `With flap effects`.
- Set **Twist distribution** to `Linear`.

Programmatic Use

Block Parameter: `theta1`
Type: character vector
Values: `'-0.1361'` | positive scalar
Default: `'-0.1361'`

Blade tip angle (rad) — Blade tip pitch angle

`0.06` (default) | nonzero positive scalar

Blade tip pitch angle θ_{tip} for ideal twist distribution, specified as a nonzero positive scalar.

Dependencies

To enable this parameter:

- Set **Modeling** to `With flap effects`.
- Set **Twist distribution** to `Ideal`.

Programmatic Use

Block Parameter: `thetaTip`
Type: character vector
Values: `'0.06'` | nonzero positive scalar
Default: `'0.06'`

Algorithms

Force and Moment

Force and moment calculated using these equations:

$$F\_xyz = \left[0;\ 0;\ -b\omega^2\right]$$
$$M\_xyz = \left[0;\ 0;\ k\omega^2\right]$$

where:

- ω is the rotor in revolutions per minute (RPM).
- *b* and *k* are experimental values:

  $$b = C_T\rho(\pi R^2)R^2$$
  $$k = -C_Q\rho(\pi R^2)R^3$$

  where:

  - *C_T* is the thrust coefficient
  - *C_Q* is the torque coefficient

Thrust Coefficient and Torque Coefficient Calculations

When the Compute CT and CQ check box is selected, the block calculates the thrust coefficient and torque coefficient using these equations.

With the inclusion of Prandtl’s tip loss function, the incremental thrust coefficient using blade element momentum theory is:

$$dC_T = 4F\lambda^2 r\ dr$$

where F is the correction factor:

$$F = \frac{2}{\pi}\cos^{-1}exp - f$$
$$f = \frac{1}{2}\frac{Nb(1 - r)}{r\varphi}$$
$$\varphi = \frac{\lambda}{r}$$

Using blade element theory, the incremental thrust coefficient is:

$$dC_T = \frac{1}{2}\sigma c_{l\alpha}(\theta(r)r - \lambda)rdr$$

with ideal twist,

$$\theta(r) = \frac{\theta_t}{r}$$

Resulting in:

$$dC_T = .5\sigma c_{l\alpha}(\theta_{t\,-\,\lambda r})rdr$$

Equating the two equations for dC_T, we have:

$$4F\lambda^2 = .5\sigma c_{l\alpha}(\theta_t - \lambda)$$

The equation can be solved to obtain λ using the `fsolve` from the Optimization Toolbox. To remove the dependency on the toolbox, the Newton-Raphson method is used:

$$fun = @(\lambda)4F(r,\lambda)\lambda^2 - .5\sigma c_{l\alpha}(\theta_t - \lambda)$$

The differential of *fun* as `diff`(*fun*,λ):

$$dfun = \frac{c_{l\alpha}\sigma}{2} + \frac{5734161139222659 * \lambda * \cos^{-1}(\exp\left(\frac{Nb*(r-1)}{2*\lambda}\right)}{1125899906842624}$$
$$+\frac{5734161139222659 * \mathrm{N}_b * \exp\left(\frac{Nb*(r-1)}{2*\lambda}\right) * (r-1)}{(4503599627370496*(1\text{-}\exp\left(\frac{Nb*(r-1)}{\lambda}\right))^{\left(\frac{1}{2}\right)})}$$

To avoid numerical issues, the ratios are approximated as:

$$dfun = \frac{c_{l\alpha}}{2} + 5.0930 * \lambda * \cos^{-1}\left(\exp\left(\frac{Nb*(r-1)}{2*\lambda}\right)\right)$$
$$+\frac{1.2732 * Nb * \exp\left(\frac{Nb*(r-1)}{2*\lambda} * (r-1)\right)}{(1 - \exp\left(\frac{Nb*(r-1)}{\lambda}\right)^{\left(\frac{1}{2}\right)})}$$

After λ is calculated, C_T, C_Q are calculated by summing up $dC_T = 4\lambda^2 r\, dr$ and $dC_Q = \lambda dC_T$ across the radius.

Version History

Introduced in R2023a

References

[1] Pounds, P. E. I. (2007). *Design, construction and control of a large quadrotor micro air vehicle* (Doctoral dissertation, Australian National University).

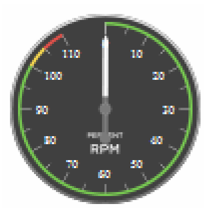
[2] Riether, F. (2016). *Agile quadrotor maneuvering using tensor-decomposition-based globally optimal control and onboard visual-inertial estimation* (Doctoral dissertation, Massachusetts Institute of Technology).

See Also

Multirotor

Revolutions Per Minute (RPM) Indicator

Display measurements for engine revolutions per minute (RPM) in percentage of RPM



Libraries:
Aerospace Blockset / Flight Instruments

Description

The RPM Indicator block displays measurements for engine revolutions per minute in percentage of RPM.

The range of values for RPM goes from 0 to 110 %. Minor ticks represent increments of 5 % RPM and major ticks represent increments of 10 % RPM.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name

Connect to signal for display, selected from list of signal names.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Scale Colors — Ranges of color bands
0 (default) | double | scalar

Ranges of color bands on the outside of the scale, specified as a finite double, or scalar value. Specify the minimum and maximum color range to display on the gauge.

To add a new color, click `+`. To remove a color, click `-`.

Programmatic Use
Block Parameter: `ScaleColors`
Type: *n*-by-1 struct array
Values: struct array with elements `Min`, `Max`, and `Color`

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is ignored for code generation.

See Also

Airspeed Indicator | Altimeter | Artificial Horizon | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Heading Indicator | Turn Coordinator

Topics

"Display Measurements with Cockpit Instruments" on page 2-59
"Programmatically Interact with Gauge Band Colors" on page 2-61
"Flight Instrument Gauges" on page 2-58

Self-Conditioned [A,B,C,D]

Implement state-space controller in self-conditioned form

Libraries:
Aerospace Blockset / GNC / Control

Description

The Self-Conditioned [A,B,C,D] block can be used to implement the state-space controller defined by

$$\begin{bmatrix} \dot{x} = Ax + Be \\ u = Cx + De \end{bmatrix}$$

in the self-conditioned form

$$\dot{z} = (A - HC)z + (B - HD)e + Hu_{meas}$$
$$u_{dem} = Cz + De$$

The input u_{meas} is a vector of the achieved actuator positions, and the output u_{dem} is the vector of controller actuator demands. In the case that the actuators are not limited, then $u_{meas} = u_{dem}$ and substituting the output equation into the state equation returns the nominal controller. In the case that they are not equal, the dynamics of the controller are set by the poles of A-HC.

Hence H must be chosen to make the poles sufficiently fast to track u_{meas} but at the same time not so fast that noise on e is propagated to u_{dem}. The matrix H is designed by a callback to the Control System Toolbox command `place` to place the poles at defined locations.

Limitations

This block requires the Control System Toolbox license.

Ports

Input

e — Control error
vector

Control error, specified as a vector.

Data Types: `double`

u_meas — Achieved actuator positions
vector

Achieved actuator positions, specified as a vector.

Data Types: `double`

Output

u_dem — Actuator demands
vector

Actuator demands, specified as a vector.

Data Types: `double`

Parameters

A-matrix — *A*-matrix of the state-space implementation

`[-1 -0.2;0 -3]` (default) | array

A-matrix of the state-space implementation. The *A*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *A*-matrix corresponding to the first entry of *v* is the identity matrix, then `A(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `Ak`
Type: character vector
Values: vector
Default: `'[-1 -0.2;0 -3]'`

B-matrix — *B*-matrix of the state-space implementation

`[1;1]` (default) | array

B-matrix of the state-space implementation, specified as an array. The *B*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *B*-matrix corresponding to the first entry of *v* is the identity matrix, then `B(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `Bk`
Type: character vector
Values: vector
Default: `'[1;1]'`

C-matrix — *C*-matrix of the state-space implementation

`[1 0]` (default) | array

C-matrix of the state-space implementation, specified as an array. The *C*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *C*-matrix corresponding to the first entry of *v* is the identity matrix, then `C(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `Ck`
Type: character vector
Values: vector
Default: `'[1 0]'`

D-matrix — *D*-matrix of the state-space implementation

`0.02` (default) | array | scalar

D-matrix of the state-space implementation. The *D*-matrix should have three dimensions, the last one corresponding to the scheduling variable *v*. For example, if the *D*-matrix corresponding to the first entry of *v* is the identity matrix, then `D(:,:,1) = [1 0;0 1];`.

Programmatic Use
Block Parameter: `Dk`
Type: character vector
Values: vector
Default: `'0.02'`

Initial state, x_initial — Initial states

0 (default) | vector

Initial states for the controller, that is, initial values for the state vector, *z*. It should have length equal to the size of the first dimension of *A*.

Programmatic Use
Block Parameter: `x_initial`
Type: character vector
Values: vector
Default: `'0'`

Poles of A-H*C — Desired poles

`[-5 -2]` (default) | vector

Desired poles of *A-H*C*, specified as a vector. Hence the number of pole locations defined should be equal to the dimension of the *A*-matrix.

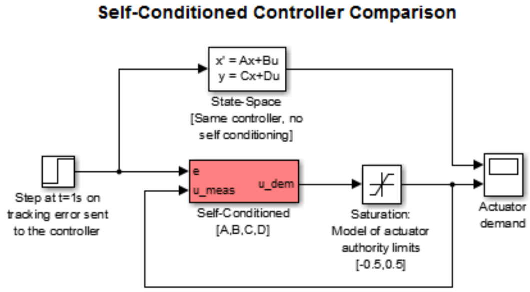
Programmatic Use
Block Parameter: `vec_w`
Type: character vector
Values: vector
Default: `'[-5 -2]'`

More About

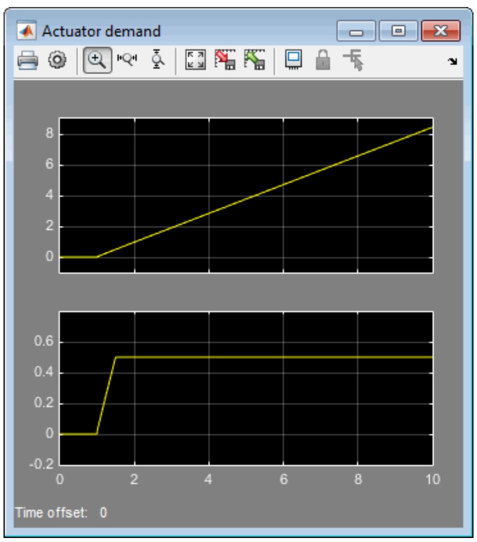
State-Space Controller

State-space controller implemented in both self-conditioned and standard state-space forms.

This Simulink model shows a state-space controller implemented in both self-conditioned and standard state-space forms. The actuator authority limits of ±0.5 units are modeled by the Saturation block.



Notice that the *A*-matrix has a zero in the 1,1 element, indicating integral action.



The top trace shows the conventional state-space implementation. The output of the controller winds up well past the actuator upper authority limit of +0.5. The lower trace shows that the self-conditioned form results in an actuator demand that tracks the upper authority limit, which means that when the sign of the control error, e, is reversed, the actuator demand responds immediately.

Version History

Introduced before R2006a

References

[1] Kautsky, Nichols, and Van Dooren, "Robust Pole Assignment in Linear State Feedback," *International Journal of Control*, Vol. 41, Number 5, 1985, pp. 1129-1155.

Extended Capabilities

C/C++ Code Generation

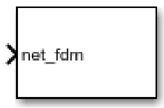
Generate C and C++ code using Simulink® Coder™.

See Also

1D Self-Conditioned [A(v),B(v),C(v),D(v)] | 2D Self-Conditioned [A(v),B(v),C(v),D(v)] | 3D Self-Conditioned [A(v),B(v),C(v),D(v)] | Saturation | Nonlinear Second-Order Actuator | Linear Second-Order Actuator

Send net_fdm Packet to FlightGear

Transmit `net_fdm` packet to destination IP address and port for FlightGear session



Libraries:
Aerospace Blockset / Animation / Flight Simulator Interfaces

Description

The Send net_fdm Packet to FlightGear block transmits the `net_fdm` packet to FlightGear on the current computer, or a remote computer on the network. The packet is constructed using the Pack net_fdm Packet for FlightGear block. The destination port should be an unused port that you can use when you launch FlightGear with the FlightGear command line flag:

```
--fdm=network,localhost,5501,5502,5503
```

This block does not produce deployable code.

Ports

Input

net_fdm — FlightGear `net_fdm` data packet
scalar

FlightGear `net_fdm` data packet, specified as a scalar.

Data Types: `uint8`

Parameters

Destination IP address — Destination IP address for remote computer

`127.0.0.1` (default) | scalar

Destination IP address, specified as a scalar.

You can use one of several techniques to determine the destination IP address, such as:

- Use 127.0.0.1 for the local computer
- Ping another computer from a Windows `cmd.exe` (or UNIX shell) prompt:

```
C:\> ping andyspc

Pinging andyspc [144.213.175.92] with 32 bytes of data:

Reply from 144.213.175.92: bytes=32 time=30ms TTL=253
Reply from 144.213.175.92: bytes=32 time=20ms TTL=253
Reply from 144.213.175.92: bytes=32 time=20ms TTL=253
Reply from 144.213.175.92: bytes=32 time=20ms TTL=253
```

```
Ping statistics for 144.213.175.92:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 20ms, Maximum =  30ms, Average =  22ms
```

- On a Windows machine, type `ipconfig` and use the returned *`IP Address`*:

```
H:\>ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . :
        IP Address. . . . . . . . . . . . : 192.168.42.178
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 192.168.42.254
```

Programmatic Use
Block Parameter: `DestinationIpAddress`
Type: character vector
Values: scalar
Default: `127.0.0.1`

Destination port — Destination port for remote computer

5502 (default) | scalar

Destination port, specified as a scalar

Programmatic Use
Block Parameter: `DestinationPort`
Type: character vector
Values: scalar
Default: `5502`

Sample time — Sample time

`1/30` (default) | scalar

Sample time (-1 for inherited), specified as a scalar.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `1/30`

Version History

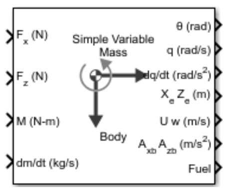
Introduced before R2006a

See Also

FlightGear Preconfigured 6DoF Animation | Generate Run Script | Pack net_fdm Packet for FlightGear | Receive net_ctrl Packet from FlightGear | Unpack net_ctrl Packet from FlightGear

Simple Variable Mass 3DOF (Body Axes)

Implement three-degrees-of-freedom equations of motion of simple variable mass with respect to body axes



Libraries:
Aerospace Blockset / Equations of Motion / 3DOF

Description

The Simple Variable Mass 3DOF (Body Axes) block implements three-degrees-of-freedom equations of motion of simple variable mass with respect to body axes. It considers the rotation in the vertical plane of a body-fixed coordinate frame about a flat Earth reference frame. For more information about the rotation and equations of motion, see “Algorithms” on page 5-782.

Ports

Input

F_x — Applied force along *x*-axis
scalar

Applied force along the body *x*-axis, specified as a scalar, in the units selected in **Units**.

Data Types: `double`

F_z — Applied force along *z*-axis
scalar

Applied force along the body *z*-axis, specified as a scalar.

Data Types: `double`

M — Applied pitching moment
scalar

Applied pitching moment, specified as a scalar.

Data Types: `double`

dm/dt — Rate of change of mass
scalar

Rate of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

g — Gravity
scalar

Gravity, specified as a scalar.

Dependencies

To enable this port, set **Gravity source** to `External`.

Data Types: `double`

V_{re} — Relative velocity
two-element vector

Relative velocity at which mass is accreted to or ablated from the body in body-fixed axes, specified as a two-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

θ — Pitch altitude
scalar

Pitch attitude, within ±pi, returned as a scalar, in radians.

Data Types: `double`

q — Pitch angular rate
scalar

Pitch angular rate, returned as a scalar, in radians per second.

Data Types: `double`

dq/dt — Pitch angular acceleration
scalar

Pitch angular acceleration, returned as a scalar, in radians per second squared.

Data Types: `double`

X_eZ_e — Location of body
two-element vector

Location of the body in the flat Earth reference frame, (*Xe, Ze*), returned as a two-element vector.

Data Types: `double`

U w — Velocity of body
two-element vector

Velocity of the body resolved into the body-fixed coordinate frame, (*u, w*), returned as a two-element vector.

Data Types: `double`

$A_{xb}A_{zb}$ — Acceleration of body
two-element vector

Acceleration of the body with respect to the body-fixed coordinate frame, (*Ax*, *Az*), returned as a two-element vector.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Dependencies

To enable this port, set **Mass type** to `Simple Variable`.

Data Types: `double`

$A_{xe}A_{ze}$ — Acceleration of body
two-element vector

Accelerations of the body with respect to the inertial (flat Earth) coordinate frame, returned as a two-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this port, select the **Include inertial acceleration** check box.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Axes — Body or wind axes

`Body` (default) | `Wind`

Body or wind axes, specified as `Wind` or `Body`.

Programmatic Use
Block Parameter: `axes`
Type: character vector
Values: `Wind` | `Body`
Default: `Body`

Mass type — Mass type

`Simple Variable` (default) | `Fixed` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 3DOF (Body Axes) • 3DOF (Wind Axes)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 3DOF (Body Axes) • Simple Variable Mass 3DOF (Wind Axes)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 3DOF (Body Axes) • Custom Variable Mass 3DOF (Wind Axes)

The `Simple Variable` selection conforms to the equations of motion described in “Algorithms” on page 5-782.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Simple Variable'`

Initial velocity — Initial velocity of body

`100` (default) | scalar

Initial velocity of the body, (V_0), specified as a scalar.

Programmatic Use
Block Parameter: `v_ini`
Type: character vector
Values: `'100'` | scalar
Default: `'100'`

Initial body attitude — Initial pitch altitude

`0` (default) | scalar

Initial pitch attitude of the body, (θ_0), specified as a scalar.

Programmatic Use
Block Parameter: `theta_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial body rotation rate — Initial pitch rotation rate

`0` (default) | scalar

Initial pitch rotation rate, (q_0), specified as a scalar.

Programmatic Use
Block Parameter: `q_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial incidence — Initial angle

`0` (default) | scalar

Initial angle between the velocity vector and the body, (α_0), specified as a scalar.

Programmatic Use
Block Parameter: `alpha_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial position (x,z) — Initial location

`[0 0]` (default) | two-element vector

Initial location of the body in the flat Earth reference frame, specified as a two-element vector.

Programmatic Use
Block Parameter: `pos_ini`
Type: character vector
Values: `'[0 0]'` | two-element vector
Default: `'[0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a scalar.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: `'1.0'` | scalar
Default: `'1.0'`

Empty mass — Mass of body when fuel tank is empty

`0.5` (default) | scalar

Mass of body when fuel tank is empty, specified as a scalar.

Programmatic Use
Block Parameter: `mass_e`
Type: character vector
Values: `'0.5'` | scalar
Default: `'0.5'`

Full mass — Mass of body when fuel tank is full

`3.0` (default) | scalar

Mass of body when fuel tank is full, specified as a scalar.

Programmatic Use
Block Parameter: `mass_f`
Type: character vector
Values: `'3.0'` | scalar
Default: `'3.0'`

Empty inertia — Body inertia when fuel tank is full

`0.5` (default) | scalar

Body inertia when the fuel tank is full, specified as a double scalar.

Programmatic Use
Block Parameter: `Iyy_e`
Type: character vector
Values: `'0.5'` | scalar
Default: `'0.5'`

Full inertia — Full inertia

`3.0` (default) | scalar

Full inertia of the body, specified as a scalar.

Programmatic Use
Block Parameter: `Iyy_f`
Type: character vector
Values: `'3.0'` | scalar

Default: '3.0'

Gravity Source — Gravity source

Internal (default) | External

Gravity source, specified as:

External	Variable gravity input to block
Internal	Constant gravity specified in mask

Programmatic Use
Block Parameter: g_in
Type: character vector
Values: 'Internal' | 'External'
Default: 'Internal'

Acceleration due to gravity — Gravity source

9.81 (default) | scalar

Acceleration due to gravity, specified as a double scalar and used if internal gravity source is selected. If gravity is to be neglected in the simulation, this value can be set to 0.

Dependencies

- To enable this parameter, set **Gravity Source** to Internal.

Programmatic Use
Block Parameter: g
Type: character vector
Values: '9.81' | scalar
Default: '9.81'

Include mass flow relative velocity — Mass flow relative velocity port

off (default) | on

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: vre_flag
Type: character vector
Values: off | on
Default: 'off'

Limit mass flow when mass is empty or full — Limit mass flow

on (default) | off

Select this check box to limit the input mass flow rate when one of these is true:

- Fuel tank is full and input mass flow rate is positive.

- Fuel tank is empty and input mass flow rate is negative.

When the input mass flow rate might cause the mass to exceed its limits, the block uses a zero mass flow rate value in the equations of motion. For more information, see Algorithms.

If you do not want to limit the input mass flow rate, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple Variable`.

Programmatic Use
Block Parameter: `mdot_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Data Types: `double`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration in flat Earth frame output port. You typically connect this signal to the accelerometer.

Dependencies

To enable the $\mathbf{A_{xe}A_{ze}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- The number of names must match the number of states, as shown for each item, or be empty. Set all or none of the block states.
- To assign names to single-variable states, enter unique names between quotes, for example, `'q'` or `"q"`.
- To assign names to two-variable states, enter a comma-separated list surrounded by braces, for example, `{'Xe','Ze'}`.
- If a state parameter is empty (`' '`), no name is assigned.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array of character vectors, or string.

Velocity: e.g., {'u, 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `vel_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Pitch attitude: e.g., 'theta' — Pitch attitude state name

`''` (default)

Pitch attitude state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `theta_statename`
Type: character vector | string
Values: `''`
Default: `''`

Position: e.g., {'Xe', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pos_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Pitch angular rate e.g., 'q' — Pitch angular rate state name

`''` (default)

Pitch angular rate state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `q_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Mass: e.g., 'mass' — Mass state name

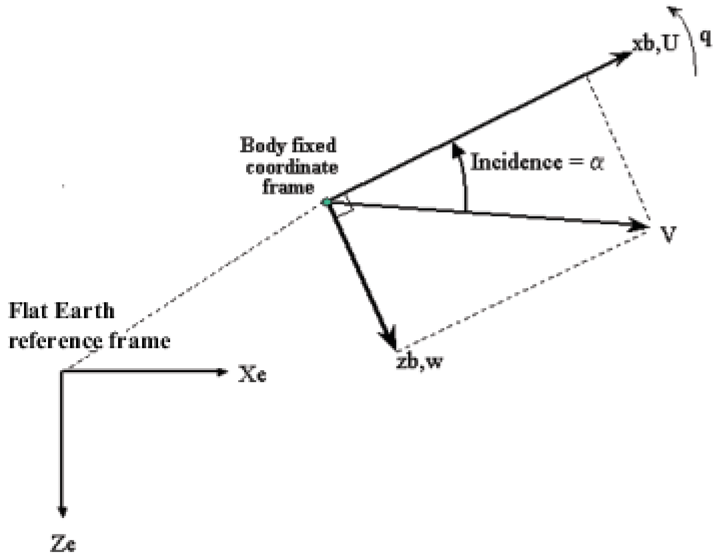
`''` (default) | scalar

Mass state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Algorithms

It considers the rotation in the vertical plane of a body-fixed coordinate frame about a flat Earth reference frame.



The equations of motion are

$$A_{xb} = \dot{u} = A_{xe} - qw$$

$$A_{zb} = \dot{w} = A_{ze} + qu$$

$$A_{xe} = \frac{(F_x - \dot{m}u_{re})}{m} - g\sin\theta$$

$$A_{ze} = \frac{(F_z - \dot{m}w_{re})}{m} + g\cos\theta$$

$$\dot{X}_e = u\cos\theta + w\sin\theta$$

$$\dot{Z}_e = -u\sin\theta + w\cos\theta$$

$$\dot{q} = \frac{M_y - \dot{I}_{yy}q}{I_{yy}}$$

$$\dot{\theta} = q$$

$$\dot{I}_{yy} = \frac{I_{yy\_full} - I_{yy\_empty}}{m_{full} - m_{empty}}\dot{m}$$

$$I_{yy} = I_{yy\_empty} + \left(I_{yy\_full} - I_{yy\_empty}\right)\frac{m - m_{empty}}{m_{full} - m_{empty}}$$

where the applied forces are assumed to act at the center of gravity of the body. Input variables are F_x, F_z, M_y, $\dot{m}$. u_{re}, w_{re}, and g are optional input variables. Mass m is limited to between m_{empty} and m_{full}. Whenever mass is saturated at empty or full, for consistency, limit $\dot{m}$ within the equations of motion.

Version History

Introduced in R2006a

The 3DOF equations of motion have been updated. Existing models created prior to R2021b that contain 3DOF equations of motion blocks continue to run. If you replace a pre-R2021b version of a 3DOF equation of motion block with an R2021b or later version, your updated model might have a higher tendency for algebraic loops. For an example of how to remove algebraic loops using unit delays, see “Remove Algebraic Loops”. For further information about algebraic loops, see “Identify Algebraic Loops in Your Model”.

Extended Capabilities

C/C++ Code Generation

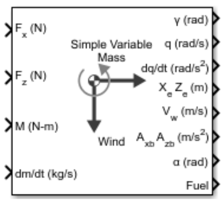
Generate C and C++ code using Simulink® Coder™.

See Also

3DOF (Body Axes) | 3DOF (Wind Axes) | 4th Order Point Mass (Longitudinal) | Custom Variable Mass 3DOF (Body Axes) | Custom Variable Mass 3DOF (Wind Axes) | Simple Variable Mass 3DOF (Wind Axes)

Simple Variable Mass 3DOF (Wind Axes)

Implement three-degrees-of-freedom equations of motion of simple variable mass with respect to wind axes



Libraries:
Aerospace Blockset / Equations of Motion / 3DOF

Description

The Simple Variable Mass 3DOF (Wind Axes) block implements three-degrees-of-freedom equations of motion of simple variable mass with respect to wind axes. The block considers the rotation in the vertical plane of a wind-fixed coordinate frame about a flat Earth reference frame. For more information about the rotation and equations of motion, see “Algorithms” on page 5-793.

Ports

Input

F_x — Applied force along wind *x*-axis
scalar

Applied force along the wind *x*-axis, specified as a scalar, in the units selected in **Units**.

Data Types: `double`

F_z — Applied force along wind *z*-axis
scalar

Applied force along the wind *z*-axis, specified as a scalar.

Data Types: `double`

M — Applied pitching moment
scalar

Applied pitching moment, specified as a scalar.

Data Types: `double`

dm/dt — Rate of change of mass
scalar

Rate of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

g — Gravity
scalar

Gravity, specified as a scalar.

Dependencies

To enable this port, set **Gravity source** to `External`.

Data Types: `double`

V_{re} — Relative velocity
two-element vector

Relative velocity at which mass is accreted to or ablated from the body in body-fixed axes, specified as a two-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

γ — Flight path angle
scalar

Flight path angle, within ±pi, returned as a scalar, in radians.

Data Types: `double`

q — Pitch angular rate
scalar

Pitch angular rate, returned as a scalar, in radians per second.

Data Types: `double`

dq/dt — Pitch angular acceleration
scalar

Pitch angular acceleration, returned as a scalar, in radians per second squared.

Data Types: `double`

X_eZ_e — Location of body
two-element vector

Location of the body in the flat Earth reference frame, (*Xe, Ze*), returned as a two-element vector.

Data Types: `double`

V_w — Velocity in wind-fixed frame
two-element vector

Velocity of the body resolved into the wind-fixed coordinate frame, (*V, 0*), returned as a two-element vector.

Data Types: `double`

$A_{xb}A_{zb}$ — Acceleration of body
two-element vector

Acceleration of the body with respect to the body-fixed coordinate frame, (*Ax, Az*), returned as a two-element vector.

Data Types: `double`

α — Angle of attack
scalar

Angle of attack, returned as a scalar, in radians.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Dependencies

To enable this port, set **Mass type** to `Simple Variable`.

Data Types: `double`

$A_{xe}A_{ze}$ — Acceleration of body
two-element vector

Accelerations of the body with respect to the inertial (flat Earth) coordinate frame, returned as a two-element vector. You typically connect this signal to the accelerometer.

Dependencies

To enable this port, select the **Include inertial acceleration** check box.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
English (Velocity in ft/s)	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
English (Velocity in kts)	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Axes — Body or wind axes

`Wind` (default) | `Body`

Body or wind axes, specified as `Wind` or `Body`

Programmatic Use
Block Parameter: `axes`
Type: character vector
Values: `Wind` | `Body`
Default: `Wind`

Mass type — Mass type

`Simple Variable` (default) | `Fixed` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 3DOF (Body Axes) • 3DOF (Wind Axes)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 3DOF (Body Axes) • Simple Variable Mass 3DOF (Wind Axes)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 3DOF (Body Axes) • Custom Variable Mass 3DOF (Wind Axes)

The `Simple Variable` selection conforms to the equations of motion described in “Algorithms” on page 5-793.

Programmatic Use
Block Parameter: `mtype`
Type: character vector

Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `'Simple Variable'`

Initial airspeed — Initial speed

`100` (default) | scalar

Initial speed of the body, (V_0), specified as a scalar.

Programmatic Use
Block Parameter: `V_ini`
Type: character vector
Values: `'100'` | scalar
Default: `'100'`

Initial flight path angle — Initial flight path angle

`0` (default) | scalar

Initial flight path angle of the body, (γ_0), specified as a scalar.

Programmatic Use
Block Parameter: `gamma_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial body rotation rate — Initial pitch rotation rate

`0` (default) | scalar

Initial pitch rotation rate, (q_0), specified as a scalar.

Programmatic Use
Block Parameter: `q_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial incidence — Initial angle

`0` (default) | scalar

Initial angle between the velocity vector and the body, (α_0), specified as a scalar.

Programmatic Use
Block Parameter: `alpha_ini`
Type: character vector
Values: `'0'` | scalar
Default: `'0'`

Initial position (x,z) — Initial location

`[0 0]` (default) | two-element vector

Initial location of the body in the flat Earth reference frame, specified as a two-element vector.

Programmatic Use
Block Parameter: `pos_ini`
Type: character vector
Values: `'[0 0]'` | two-element vector
Default: `'[0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a scalar.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: `'1.0'` | scalar
Default: `'1.0'`

Empty mass — Mass of body when fuel tank is empty

`0.5` (default) | scalar

Mass of body when fuel tank is empty, specified as a scalar.

Programmatic Use
Block Parameter: `mass_e`
Type: character vector
Values: `'0.5'` | scalar
Default: `'0.5'`

Full mass — Mass of body when fuel tank is full

`3.0` (default) | scalar

Mass of body when fuel tank is full, specified as a scalar.

Programmatic Use
Block Parameter: `mass_f`
Type: character vector
Values: `'3.0'` | scalar
Default: `'3.0'`

Empty inertia body axes — Inertia of body when fuel tank is empty

`0.5` (default) | scalar

Inertia of body when fuel tank is empty, specified as a scalar.

Dependencies

To enable this parameter, set **Mass type** to `Simple Variable`.

Programmatic Use
Block Parameter: `Iyy_e`
Type: character vector
Values: `'1.0'` | scalar

Default: `'1.0'`

Full inertia body axes — Body inertia when fuel tank is full

`3.0` (default) | scalar

Body inertia when the fuel tank is full, specified as a scalar.

Dependencies

To enable this parameter, set **Mass type** to `Simple Variable`.

Programmatic Use

Block Parameter: `Iyy_f`
Type: character vector
Values: `'3.0'` | scalar
Default: `'3.0'`

Gravity Source — Gravity source

`Internal` (default) | `External`

Gravity source, specified as:

`External`	Variable gravity input to block
`Internal`	Constant gravity specified in mask

Programmatic Use

Block Parameter: `g_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Acceleration due to gravity — Gravity source

`9.81` (default) | scalar

Acceleration due to gravity, specified as a double scalar and used if internal gravity source is selected. If gravity is to be neglected in the simulation, this value can be set to `0`.

Dependencies

- To enable this parameter, set **Gravity Source** to `Internal`.

Programmatic Use

Block Parameter: `g`
Type: character vector
Values: `'9.81'` | scalar
Default: `'9.81'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use

Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `'off'`

Limit mass flow when mass is empty or full — Limit mass flow

`on` (default) | `off`

Select this check box to limit the input mass flow rate when one of these is true:

- Fuel tank is full and input mass flow rate is positive.
- Fuel tank is empty and input mass flow rate is negative.

When the input mass flow rate might cause the mass to exceed its limits, the block uses a zero mass flow rate value in the equations of motion. For more information, see “Algorithms” on page 5-793.

If you do not want to limit the input mass flow rate, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple Variable`.

Programmatic Use

Block Parameter: `mdot_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Data Types: `double`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration in flat Earth frame output port. You typically connect this signal to the accelerometer.

Dependencies

To enable the $\mathbf{A_{xe}A_{ze}}$ port, select this parameter.

Programmatic Use

Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- The number of names must match the number of states, as shown for each item, or be empty. Set all or none of the block states.

- To assign names to single-variable states, enter unique names between quotes, for example, `'q'` or `"q"`.
- To assign names to two-variable states, enter a comma-separated list surrounded by braces, for example, `{'Xe','Ze'}`.
- If a state parameter is empty (`' '`), no name is assigned.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array of character vectors, or string.

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `V_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Position: e.g., {'Xe', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pos_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rate: e.g., 'q' — Body rotation state name

`''` (default) | scalar

Body rotation rate state names, specified as a character vector or string.

Programmatic Use
Block Parameter: `q_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Flight path angle: e.g., 'gamma' — Flight path angle state name

`''` (default)

Flight path angle state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `gamma_statename`
Type: character vector | string
Values: `''` | scalar

Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

`''` (default) | scalar

Incidence angle state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Mass: e.g., 'mass' — Mass state name

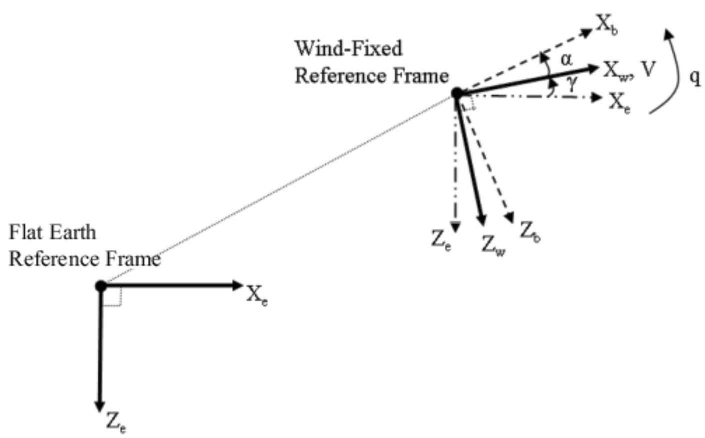
`''` (default) | scalar

Mass state name, specified as a character vector or string.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector | string
Values: `''` | scalar
Default: `''`

Algorithms

The block considers the rotation in the vertical plane of a wind-fixed coordinate frame about a flat Earth reference frame.



The equations of motion are

$$A_{xb} = A_{xe} - qV\sin\alpha$$

$$A_{zb} = A_{ze} + qV\cos\alpha$$

$$A_{xe} = \left(\frac{F_x}{m} - g\sin\gamma\right)\cos\alpha - \left(\frac{F_z}{m} + g\cos\gamma\right)\sin\alpha$$

$$A_{ze} = \left(\frac{F_x}{m} - g\sin\gamma\right)\sin\alpha + \left(\frac{F_z}{m} + g\cos\gamma\right)\cos\alpha$$

$$\dot{V} = \frac{(F_x + \dot{m}u_{re})}{m} - g\sin\gamma$$

$$\dot{X}_e = V\cos\gamma$$

$$\dot{Z}_e = -V\sin\gamma$$

$$\dot{q} = \frac{M_y - \dot{I}_{yy}q}{I_{yy}}$$

$$\dot{\gamma} = q - \dot{\alpha}$$

$$\dot{\alpha} = \frac{(F_z + \dot{m}w_{re})}{mV} + \frac{g}{V}\cos\gamma + q$$

$$\dot{I}_{yy} = \frac{I_{yy\_full} - I_{yy\_empty}}{m_{full} - m_{empty}}\dot{m}$$

$$I_{yy} = I_{yy\_empty} + \left(I_{yy\_full} - I_{yy\_empty}\right)\frac{m - m_{empty}}{m_{full} - m_{empty}}$$

where the applied forces are assumed to act at the center of gravity of the body. Input variables are wind-axes forces F_x and F_z, body moment M_y, and $\dot{m}$. u_{re}, w_{re}, and g are optional input variables. Mass m is limited between m_{empty} and m_{full}. Whenever mass is saturated at empty or full, for consistency, limit $\dot{m}$ within the equations of motion.

Version History

Introduced in R2006a

R2021b: Simple Variable Mass 3DOF (Wind Axes) Block Changes

Behavior changed in R2021b

The 3DOF equations of motion have been updated. Existing models created prior to R2021b that contain 3DOF equations of motion blocks continue to run. If you replace a pre-R2021b version of a 3DOF equation of motion block with an R2021b or later version, your updated model might have a higher tendency for algebraic loops. For an example of how to remove algebraic loops using unit delays, see “Remove Algebraic Loops”. For further information about algebraic loops, see “Identify Algebraic Loops in Your Model”.

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. Hoboken, NJ: John Wiley & Sons, 1992.

Extended Capabilities

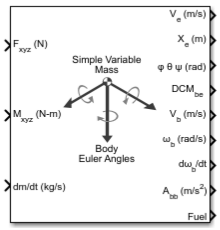
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

3DOF (Body Axes) | 3DOF (Wind Axes) | Custom Variable Mass 3DOF (Body Axes) | Custom Variable Mass 3DOF (Wind Axes) | Simple Variable Mass 3DOF (Body Axes)

Simple Variable Mass 6DOF (Euler Angles)

Implement Euler angle representation of six-degrees-of-freedom equations of motion of simple variable mass



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Simple Variable Mass 6DOF (Euler Angles) block considers the rotation of a body-fixed coordinate frame (X_b, Y_b, Z_b) about a flat Earth reference frame (X_e, Y_e, Z_e).

For a description of the coordinate system and the translational dynamics, see the description for the Simple Variable Mass 6DOF (Euler Angles) block. For more information on the body-fixed coordinate frame, see “Algorithms” on page 5-804.

Limitations

The block assumes that the applied forces are acting at the center of gravity of the body.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt (kg/s) — Rate of change of mass
scalar

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

V_{re} — Relative velocity
three-element vector

One or more relative velocities, specified as a three-element vector, at which the mass is accreted to or ablated from the body in body-fixed axes.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

φ θ ψ (rad) — Euler rotation angles
three-element vector

Euler rotation angles [roll, pitch, yaw], returned as three-element vector, in radians.

Data Types: `double`

DCM_{be} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity in body-fixed frame
three-element vector

Velocity in body-fixed frame, returned as a three-element vector.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type
`Simple Variable` (default) | `Fixed` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the equations of motion in “Algorithms” on page 5-804.

Programmatic Use
Block Parameter: mtype
Type: character vector
Values: Fixed | Simple Variable | Custom Variable
Default: Simple Variable

Representation — Equations of motion representation

Euler Angles (default) | Quaternion

Equations of motion representation, specified according to the following table.

Representation	Description
Euler Angles	Use Euler angles within equations of motion.
Quaternion	Use quaternions within equations of motion.

The Euler Angles selection conforms to the equations of motion in "Algorithms" on page 5-804.

Programmatic Use
Block Parameter: rep
Type: character vector
Values: Euler Angles | Quaternion
Default: 'Euler Angles'

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

[0 0 0] (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: xme_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial velocity in body axes [U,v,w] — Velocity in body axes

[0 0 0] (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: Vm_0
Type: character vector
Values: '[0 0 0]' | three-element vector
Default: '[0 0 0]'

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

[0 0 0] (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Empty mass — Empty mass

`0.5` (default) | scalar

Empty mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_e`
Type: character vector
Values: double scalar
Default: `'0.5'`

Full mass — Full mass of body

`2.0` (default) | scalar

Full mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_f`
Type: character vector
Values: double scalar
Default: `'2.0'`

Empty inertia matrix — Empty inertia matrix

`eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the empty inertia of the body, specified as 3-by-3 matrix.

Programmatic Use
Block Parameter: `inertia_e`
Type: character vector
Values: `'eye(3)'` | 3-by-3 matrix
Default: `'eye(3)'`

Full inertia matrix — Full inertia of body

`2*eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the full inertia of the body, specified as 3-by-3 matrix.

Programmatic Use
Block Parameter: `inertia_f`
Type: character vector
Values: `'2*eye(3)'` | 3-by-3 matrix
Default: `'2*eye(3)'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Euler rotation angles: e.g., {'phi', 'theta', 'psi'} — Euler rotation state name

`''` (default) | comma-separated list surrounded by braces

Euler rotation angle state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `eul_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Mass: e.g., 'mass' — Mass state name

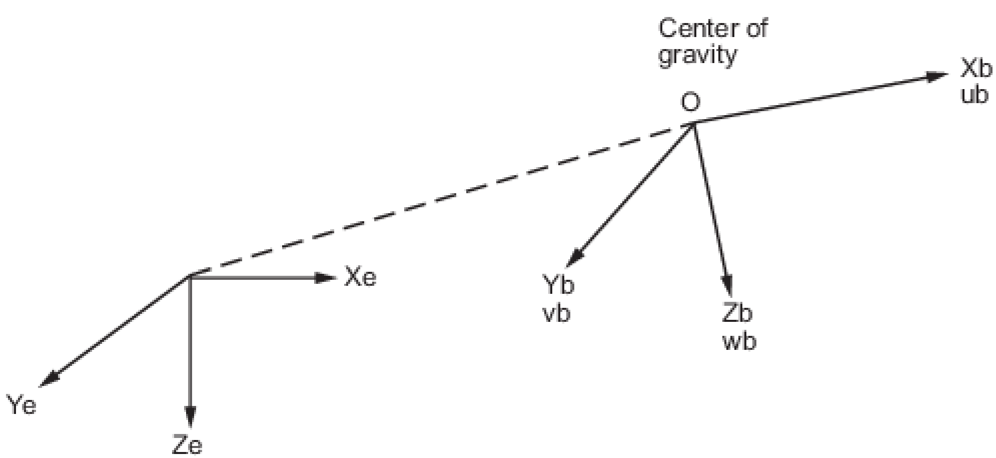
`''` (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The origin of the body-fixed coordinate frame is the center of gravity of the body, and the body is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The flat Earth reference frame is considered inertial, an excellent approximation that allows the forces due to the Earth's motion relative to the fixed stars to be neglected.



Flat Earth reference frame

The translational motion of the body-fixed coordinate frame is given below, where the applied forces $[F_x\ F_y F_z]^T$ are in the body-fixed frame. Vre_b is the relative velocity in the body axes at which the mass flow ($\dot{m}$) is ejected or added to the body in body axes.

$$\bar{F}_b = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m(\dot{\bar{V}}_b + \bar{\omega} \times \bar{V}_b) + \dot{m}\bar{V}re_b$$

$$A_{be} = \frac{\bar{F}_b - \dot{m}\bar{V}_{re}}{m}$$

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{\omega}_b \end{bmatrix} = \frac{\bar{F}_b - \dot{m}\bar{V}_{re}}{m} - \bar{\omega} \times \bar{V}_b$$

$$\bar{V}_b = \begin{bmatrix} u_b \\ v_b \\ w_b \end{bmatrix}, \bar{\omega} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

The rotational dynamics of the body-fixed frame are given below, where the applied moments are [L M N]^T, and the inertia tensor I is with respect to the origin O.

$$\overline{M}_B = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\bar{\omega}} + \bar{\omega} \times (I\bar{\omega}) + \dot{I}\,\bar{\omega}$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The inertia tensor is determined using a table lookup which linearly interpolates between I_{full} and I_{empty} based on mass (m). While the rate of change of the inertia tensor is estimated by the following equation.

$$\dot{I} = \frac{I_{full} - I_{empty}}{m_{full} - m_{empty}}\dot{m}$$

The relationship between the body-fixed angular velocity vector, [p q r]^T, and the rate of change of the Euler angles, $[\dot{\phi}\dot{\theta}\dot{\psi}]^T$, can be determined by resolving the Euler rates into the body-fixed coordinate frame.

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}\begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \equiv J^{-1}\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

Inverting J then gives the required relationship to determine the Euler rate vector.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = J\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & (\sin\phi\tan\theta) & (\cos\phi\tan\theta) \\ 0 & \cos\phi & -\sin\phi \\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta} \end{bmatrix}\begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

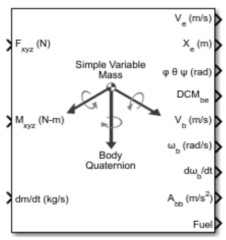
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Simple Variable Mass 6DOF (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion of simple variable mass with respect to body axes



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Simple Variable Mass 6DOF (Quaternion) implements a quaternion representation of six-degrees-of-freedom equations of motion of simple variable mass with respect to body axes.

For a description of the coordinate system and the translational dynamics, see the description for the Simple Variable Mass 6DOF (Euler Angles) block. Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For more information on the integration of the rate of change of the quaternion vector, see “Algorithms” on page 5-815.

Limitations

The block assumes that the applied forces are acting at the center of gravity of the body.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt (kg/s) — Rate of change of mass
scalar

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

V_{re} — Relative velocity
three-element vector

One or more relative velocities, specified as a three-element vector, at which the mass is accreted to or ablated from the body in body-fixed axes.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

φ θ ψ (rad) — Euler rotation angles
three-element vector

Euler rotation angles [roll, pitch, yaw], returned as three-element vector, in radians.

Data Types: `double`

DCM_{be} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity in body-fixed frame
three-element vector

Velocity in body-fixed frame, returned as a three-element vector.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
English (Velocity in kts)	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: units
Type: character vector
Values: Metric (MKS) | English (Velocity in ft/s) | English (Velocity in kts)
Default: Metric (MKS)

Mass Type — Mass type

Simple Variable (default) | Fixed | Custom Variable

Mass type, specified according to the following table.

Mass Type	Description	Default For
Fixed	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
Simple Variable	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
Custom Variable	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The Simple Variable selection conforms to the equations of motion in “Algorithms” on page 5-815.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Quaternion` (default) | `Euler Angles`

Equations of motion representation, specified according to the following table.

Representation	Description
`Quaternion`	Use quaternions within equations of motion.
`Euler Angles`	Use Euler angles within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-815.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Euler Angles` | `Quaternion`
Default: `'Euler Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial velocity in body axes [U,v,w] — Velocity in body axes

`[0 0 0]` (default) | three-element vector

Initial velocity in body axes, specified as a three-element vector, in the body-fixed coordinate frame.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

`[0 0 0]` (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Empty mass — Empty mass

`0.5` (default) | scalar

Empty mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_e`
Type: character vector
Values: double scalar
Default: `'0.5'`

Full mass — Full mass of body

`2.0` (default) | scalar

Full mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_f`
Type: character vector
Values: double scalar
Default: `'2.0'`

Empty inertia matrix — Empty inertia matrix

`eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the empty inertia of the body, specified as 3-by-3 matrix.

Programmatic Use
Block Parameter: `inertia_e`
Type: character vector
Values: `'eye(3)'` | 3-by-3 matrix
Default: `'eye(3)'`

Full inertia matrix — Full inertia of body

`2*eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the full inertia of the body, specified as 3-by-3 matrix.

Programmatic Use
Block Parameter: `inertia_f`
Type: character vector
Values: `'2*eye(3)'` | 3-by-3 matrix
Default: `'2*eye(3)'`

Gain for quaternion normalization — Gain

`1.0` (default) | scalar

Gain to maintain the norm of the quaternion vector equal to 1.0, specified as a double scalar.

Programmatic Use
Block Parameter: `k_quat`
Type: character vector
Values: `1.0` | double scalar
Default: `1.0`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Mass: e.g., 'mass' — Mass state name

`''` (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The equation of the integration of the rate of change of the quaternion vector follows. The gain K drives the norm of the quaternion state vector to 1.0 should ε become nonzero. You must choose the value of this gain with care, because a large value improves the decay rate of the error in the norm, but also slows the simulation because fast dynamics are introduced. An error in the magnitude in one element of the quaternion vector is spread equally among all the elements, potentially increasing the error in the state vector.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = {}^1\!/_2 \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} + K\varepsilon \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

$$\varepsilon = 1 - (q_0{}^2 + q_1{}^2 + q_3{}^2 + q_4{}^2)$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*. 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

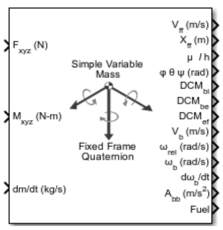
See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Topics
"About Aerospace Coordinate Systems" on page 2-8

Simple Variable Mass 6DOF ECEF (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion of simple variable mass in Earth-centered Earth-fixed (ECEF) coordinates



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Simple Variable Mass 6DOF ECEF (Quaternion) block implements a quaternion representation of six-degrees-of-freedom equations of motion of simple variable mass in Earth-centered Earth-fixed (ECEF) coordinates. It considers the rotation of an Earth-centered Earth-fixed (ECEF) coordinate frame (X_{ECEF}, Y_{ECEF}, Z_{ECEF}) about an Earth-centered inertial (ECI) reference frame (X_{ECI}, Y_{ECI}, Z_{ECI}). The origin of the ECEF coordinate frame is the center of the Earth. For more information on the ECEF coordinate frame, see "Algorithms" on page 5-830.

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

Limitations

- This implementation assumes that the applied forces are acting at the center of gravity of the body.
- This implementation generates a geodetic latitude that lies between ±90 degrees, and longitude that lies between ±180 degrees. Additionally, the MSL altitude is approximate.
- The Earth is assumed to be ellipsoidal. By setting flattening to 0.0, a spherical planet can be achieved. The Earth's precession, nutation, and polar motion are neglected. The celestial longitude of Greenwich is Greenwich Mean Sidereal Time (GMST) and provides a rough approximation to the sidereal time.
- The implementation of the ECEF coordinate system assumes that the origin is at the center of the planet, the *x*-axis intersects the Greenwich meridian and the equator, the *z*-axis is the mean spin axis of the planet, positive to the north, and the *y*-axis completes the right-hand system.
- The implementation of the ECI coordinate system assumes that the origin is at the center of the planet, the *x*-axis is the continuation of the line from the center of the Earth toward the vernal equinox, the *z*-axis points in the direction of the mean equatorial plane's north pole, positive to the north, and the *y*-axis completes the right-hand system.

Ports

Input

F_{xyz} — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz} — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt — Rates of change of mass
three-element vector

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a three-element vector.

Data Types: `double`

$L_G(0)$ — Initial celestial longitude of Greenwich
scalar

Greenwich meridian initial celestial longitude angle, specified as a scalar.

Dependencies

To enable this port

- Set **Celestial longitude of Greenwich** to `External`.
- set **Planet model** to `Earth`.

Data Types: `double`

$L_{PM}(0)$ — Prime meridian initial celestial longitude angle
scalar

Prime meridian initial celestial longitude angle, specified as a scalar.

Dependencies

To enable this port

- Set **Celestial longitude of prime meridian** to `External`.
- Set **Planet model** to `Custom`.

Data Types: `double`

V_{re} — Relative velocities
three-element vector

One or more relative velocities at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a three-element vector.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_{ff} — Velocity of body with respect to ECEF frame,
three-element vector

Velocity of body with respect to ECEF frame, expressed in ECEF frame, returned as a three-element vector.

Data Types: `double`

X_{ff} — Position in ECEF reference frame
three-element vector

Position in ECEF reference frame, returned as a three-element vector.

Data Types: `double`

μ l h — Position in geodetic latitude, longitude, and altitude
three-element vector | M-by-3 array

Position in geodetic latitude, longitude, and altitude, in degrees, returned as a three-element vector or M-by-3 array, in selected units of length, respectively.

Data Types: `double`

φ θ Ψ (rad) — Body rotation angles
three-element vector

Body rotation angles [roll, pitch, yaw], returned as a three-element vector, in radians. Euler rotation angles are those between body and NED coordinate systems.

Data Types: `double`

DCM_{bi} — Coordinate transformation from ECI axes
3-by-3 matrix

Coordinate transformation from ECI axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

DCM_{be} — Coordinate transformation from NED axes to body-fixed axes
3-by-3 matrix

Coordinate transformation from NED axes to body-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

DCM_{ef} — Coordinate transformation from fixed-frame axes
3-by-3 matrix

Coordinate transformation from fixed-frame axes to NED axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_b — Velocity of body with respect to fixed-frame
three-element vector

Velocity of body with respect to fixed-frame, returned as a three-element vector.

Data Types: `double`

ω_{rel} — Relative angular rates of body with respect to NED frame
three-element vector

Relative angular rates of body with respect to NED frame, expressed in body frame and returned as a three-element vector, in radians per second.

Data Types: `double`

ω_b — Angular rates of body with respect to inertial frame
three-element vector

Angular rates of the body with respect to inertial frame, expressed in body frame and returned as a three-element vector, in radians per second.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations of the body with respect to inertial frame
three-element vector

Angular accelerations of the body with respect to inertial frame, expressed in body frame and returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations of the body with respect to the body-fixed axes with the body-fixed coordinate frame, returned as a three-element vector.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Data Types: `double`

$A_{b\ ff}$ — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to fixed-frame, returned as a three-element vector.

Dependencies

To enable this point, **Include inertial acceleration**.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type
`Simple Variable` (default) | `Fixed` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)

Mass Type	Description	Default For
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the equations of motion in “Algorithms” on page 5-830.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Initial position in geodetic latitude, longitude and altitude [mu,l,h] — Initial location of rigid body

`[0 0 0]` (default) | three-element vector

Initial location of the rigid body in the geodetic reference frame, specified as a three-element vector. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Programmatic Use
Block Parameter: `xg_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial velocity in body axes [U,v,w] — Velocity in body axes

`[0 0 0]` (default) | three-element vector

Initial velocity of the body with respect to the ECEF frame, expressed in the body frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial Euler orientation [roll, pitch, yaw] — Initial Euler orientation

`[0 0 0]` (default) | three-element vector

Initial Euler orientation angles [roll, pitch, yaw], specified as a three-element vector, in radians. Euler rotation angles are those between the body and north-east-down (NED) coordinate systems.

Programmatic Use
Block Parameter: `eul_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Empty mass — Empty mass

`0.5` (default) | scalar

Empty mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_e`
Type: character vector

Values: double scalar
Default: `'0.5'`

Full mass — Full mass of body

`2.0` (default) | scalar

Full mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_f`
Type: character vector
Values: double scalar
Default: `'2.0'`

Empty inertia matrix — Empty inertia matrix

`eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the empty inertia of the body, specified as 3-by-3 matrix.

Programmatic Use
Block Parameter: `inertia_e`
Type: character vector
Values: `'eye(3)'` | 3-by-3 matrix
Default: `'eye(3)'`

Full inertia matrix — Full inertia of body

`2*eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the full inertia of the body, specified as 3-by-3 matrix.

Programmatic Use
Block Parameter: `inertia_f`
Type: character vector
Values: `'2*eye(3)'` | 3-by-3 matrix
Default: `'2*eye(3)'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use

Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

Planet

Planet model — Planet model

`Earth (WGS84)` (default) | `Custom`

Planet model to use, `Custom` or `Earth (WGS84)`.

Programmatic Use

Block Parameter: `ptype`
Type: character vector
Values: `'Earth (WGS84)'` | `'Custom'`
Default: `'Earth (WGS84)'`

Equatorial radius — Radius of planet at equator

`6378137` (default) | scalar

Radius of the planet at its equator, specified as a double scalar, in the same units as the desired units for the ECEF position.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use

Block Parameter: `R`
Type: character vector
Values: double scalar
Default: `'6378137'`

Flattening — Flattening of planet

`1/298.257223563` (default) | scalar

Flattening of the planet, specified as a double scalar.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use

Block Parameter: `F`
Type: character vector
Values: double scalar

Default: '1/298.257223563'

Rotational rate — Rotational rate

7292115e-11 (default) | scalar

Rotational rate of the planet, specified as a scalar, in rad/s.

Dependencies

To enable this parameter, set **Planet model** to Custom.

Programmatic Use
Block Parameter: w_E
Type: character vector
Values: double scalar
Default: '7292115e-11'

Celestial longitude of Greenwich source — Source of Greenwich meridian initial celestial longitude

Internal (default) | External

Source of Greenwich meridian initial celestial longitude, specified as:

Internal	Use celestial longitude value from **Celestial longitude of Greenwich**.
External	Use external input for celestial longitude value.

Dependencies

- To enable this parameter, set **Planet model** to Earth.
- Setting this parameter to External enables the $\mathbf{L_G(0)}$ port.
- If **Planet model** is set to Custom, the parameter name changes to **Celestial longitude of prime meridian source**.

Programmatic Use
Block Parameter: angle_in
Type: character vector
Values: 'Internal' | 'External'
Default: 'Internal'

Celestial longitude of Greenwich [deg] — Initial angle

0 (default) | scalar

Initial angle between Greenwich meridian and the *x*-axis of the inertial frame, specified as a double scalar.

Dependencies

- To enable this parameter, set:
  - **Celestial longitude of Greenwich source** to Internal.
  - **Planet model** to Earth (WGS84).

- If **Planet model** is set to `Custom`, the parameter name changes to **Celestial longitude of prime meridian [deg]**.

Programmatic Use
Block Parameter: `LPM0`
Type: character vector
Values: double scalar
Default: `'0'`

Celestial longitude of prime meridian source — Source of prime meridian initial celestial longitude
`Internal` (default) | `External`

Source of prime meridian initial celestial longitude, specified as:

`Internal`	Use celestial longitude value from **Celestial longitude of prime meridian**.
`External`	Use external input for celestial longitude value.

Dependencies

- To enable this parameter, set **Planet model** to `Custom`.
- Setting this parameter to `External` enables the $\mathbf{L_{PM}(0)}$ port.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich source**.

Programmatic Use
Block Parameter: `angle_in`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Celestial longitude of prime meridian [deg] — Initial angle
`0` (default) | scalar

Initial angle between prime meridian and the *x*-axis of the ECI frame, specified as a double scalar.

Dependencies

- To enable this parameter, set:
  - **Celestial longitude of prime meridian source** to `Internal`.
  - **Planet model** to `Custom`.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich [deg]**.

Programmatic Use
Block Parameter: `LPM0`
Type: character vector
Values: double scalar
Default: `'0'`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., {'U', 'v', 'w'} — Velocity state name

`''` (default) | comma-separated list surrounded by braces

Velocity state names, specified as comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces

Default: `''`

Fixed frame position: e.g., {'Xff', 'Yff', 'Zff'} — ECEF position state name

`''` (default) | comma-separated list surrounded by braces

ECEF position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `posfixedframe_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Inertial position: e.g., {'Xinertial', 'Yinertial', 'Zinertial'} — Inertial position state names

`''` (default) | comma-separated list surrounded by braces

Inertial position state names, specified as a comma-separated list surrounded by braces.

Default value is `''`.

Programmatic Use
Block Parameter: `posinertial_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Celestial longitude of Greenwich: e.g., 'LG' — Celestial longitude state name

`''` (default) | character vector

Celestial longitude of Greenwich state name, specified as a character vector.

Dependencies

- To enable this parameter, set:**Planet model** to `Earth (WGS84)`.
- If **Planet model** is set to `Custom`, the parameter name changes to **Celestial longitude of prime meridian: e.g., 'LPM'**.

Programmatic Use
Block Parameter: `LPM_statename`
Type: character vector
Values: `''` | scalar
Default: `''`

Celestial longitude of prime meridian: e.g., 'LPM' — Celestial longitude prime meridian
`''` (default) | character vector

Celestial longitude of prime meridian state name, specified as a character vector.

Dependencies

- To enable this parameter, set:**Planet model** to `Custom`.
- If **Planet model** is set to `Earth (WGS84)`, the parameter name changes to **Celestial longitude of Greenwich: e.g., 'LG'**.

Programmatic Use
Block Parameter: `LPM_statename`
Type: character vector
Values: `''` | scalar
Default: `''`

Mass: e.g., 'mass' — Mass state name

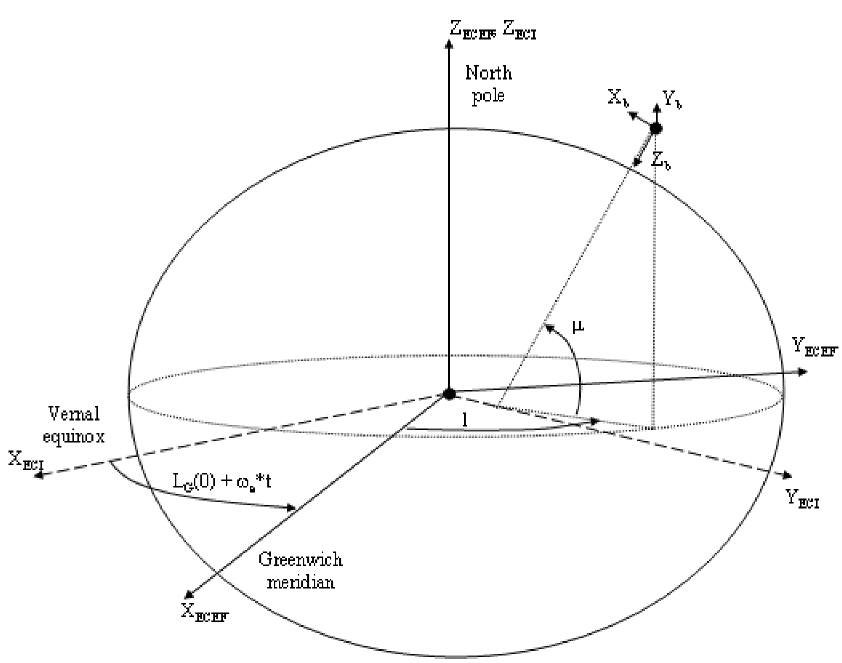
`''` (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The origin of the ECEF coordinate frame is the center of the Earth. The body of interest is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The representation of the rotation of ECEF frame from ECI frame is simplified to consider only the constant rotation of the ellipsoid Earth (ω_e) including an initial celestial longitude ($L_G(0)$). This excellent approximation allows the forces due to the Earth's complex motion relative to the “fixed stars” to be neglected.



The translational motion of the ECEF coordinate frame is given below, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the body frame. Vre_b is the relative velocity in the wind axes at which the mass flow ($\dot{m}$) is ejected or added to the body axes.

$$\bar{F}_b = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m\left(\dot{\bar{V}}_b + \bar{\omega}_b \times \bar{V}_b + DCM_{bf}\bar{\omega}_e \times \bar{V}_b + DCM_{bf}\left(\bar{\omega}_e \times \left(\bar{\omega}_e \times \bar{X}_f\right)\right)\right)$$

$$+\dot{m}\left(\bar{V}re_b + DCM_{bf}\left(\bar{\omega}_e \times \bar{X}_f\right)\right)$$

$$A_{bb} = \begin{bmatrix} \dot{u}_b \\ \dot{v}_b \\ \dot{w}_b \end{bmatrix} = \frac{\bar{F}_b - \dot{m}\left(\bar{V}_{re_b} + DCM_{bf}\left(w_e \times X_f\right)\right)}{m}$$

$$-\left[\bar{\omega}_b \times \bar{V}_b + DCM\bar{\omega}_e \times \bar{V}_b + DCM_{bf}\left(\bar{\omega}_e\left(\bar{\omega}_e \times X_f\right)\right)\right]$$

$$A_{becef} = \frac{\bar{F}_b - \dot{m}\left(\bar{V}_{re_b} + DCM_{bf}\left(\omega_e \times X_f\right)\right)}{m}$$

where the change of position in ECEF $\dot{x}_f(\dot{x}_i)$ is calculated by

$$\dot{x}_f = DCM_{fb}\bar{V}_b$$

and the velocity of the body with respect to ECEF frame, expressed in body frame ($\bar{V}_b$), angular rates of the body with respect to ECI frame, expressed in body frame ($\bar{\omega}_b$). Earth rotation rate ($\bar{\omega}_e$), and relative angular rates of the body with respect to north-east-down (NED) frame, expressed in body frame ($\bar{\omega}_{rel}$) are defined as

$$\bar{V}_b = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \bar{\omega}_{rel} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \bar{\omega}_e = \begin{bmatrix} 0 \\ 0 \\ \omega_e \end{bmatrix}$$

$$\bar{\omega}_b = \bar{\omega}_{rel} + DCM_{bf}\bar{\omega}_e + DCM_{be}\bar{\omega}_{ned}$$

$$\bar{\omega}_{ned} = \begin{bmatrix} \dot{l}\cos\mu \\ -\dot{\mu} \\ -\dot{l}\sin\mu \end{bmatrix} = \begin{bmatrix} V_E/(N+h) \\ -V_N/(M+h) \\ V_E\tan\mu/(N+h) \end{bmatrix}$$

The rotational dynamics of the body defined in body-fixed frame are given below, where the applied moments are $[L\ M\ N]^{\mathrm{T}}$, and the inertia tensor I is with respect to the origin O.

$$\bar{M}_b = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = \bar{I}\dot{\bar{\omega}}_b + \bar{\omega}_b \times (\bar{I}\bar{\omega}_b) + \dot{I}\bar{\omega}_b$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The inertia tensor is determined using a table lookup which linearly interpolates between I_{full} and I_{empty} based on mass (m). The rate of change of the inertia tensor is estimated by the following equation.

$$\dot{I} = \frac{I_{full} - I_{empty}}{m_{full} - m_{empty}}\dot{m}$$

The integration of the rate of change of the quaternion vector is given below.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -\tfrac{1}{2}\begin{bmatrix} 0 & \omega_b(1) & \omega_b(2) & \omega_b(3) \\ -\omega_b(1) & 0 & -\omega_b(3) & \omega_b(2) \\ -\omega_b(2) & \omega_b(3) & 0 & -\omega_b(1) \\ -\omega_b(3) & -\omega_b(2) & \omega_b(1) & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Version History

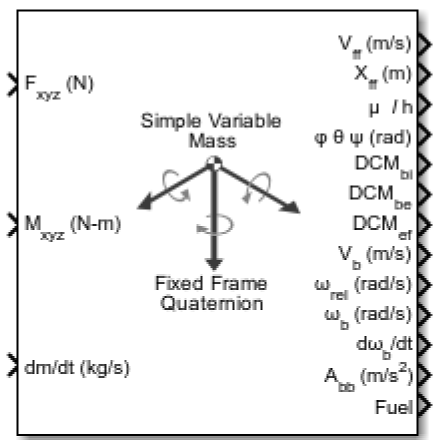
Introduced in R2006a

R2023b: Simple Variable Mass 6DOF ECEF (Quaternion) Block Coordinate Frame Changes

Behavior changed in R2023b

These updates better clarify the coordinate frame of the Simple Variable Mass 6DOF ECEF (Quaternion) block when **Central body** is `Custom`:

- New block icon.



- Port name subscripts of `ecef` have changed to `ff`.
- The **Celestial longitude of Greenwich source** programmatic name has changed from `LG0` to `LPM`. Existing scripts continue to work.
- Updated block parameter names, but models from previous releases continue to work.

Old Parameter Name or Setting	New Parameter Name
ECEF position...	**Fixed frame position**...
When **Planet model** is `Earth`	**Celestial longitude of Greenwich source** and **Celestial longitude of Greenwich [deg]**
When **Planet model** is `Custom`	**Celestial longitude of prime meridian source** and **Celestial longitude of prime meridian [deg]**

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation, 2nd ed*. Hoboken, NJ: John Wiley & Sons, 2003.

[2] McFarland, Richard E. "A Standard Kinematic Model for Flight simulation at NASA-Ames" NASA CR-2497.

[3] "Supplement to Department of Defense World Geodetic System 1984 Technical Report: Part I - Methods, Techniques and Data Used in WGS84 Development." DMA TR8350.2-A.

Extended Capabilities

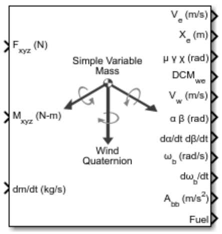
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Simple Variable Mass 6DOF Wind (Quaternion)

Implement quaternion representation of six-degrees-of-freedom equations of motion of simple variable mass with respect to wind axes



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Simple Variable Mass 6DOF Wind (Quaternion) block implements a quaternion representation of six-degrees-of-freedom equations of motion of simple variable mass with respect to wind axes. It considers the rotation of a wind-fixed coordinate frame (X_w, Y_w, Z_w) about an flat Earth reference frame (X_e, Y_e, Z_e).

Aerospace Blockset uses quaternions that are defined using the scalar-first convention. For more information on the wind-fixed coordinate frame, see “Algorithms” on page 5-843.

Limitations

The block assumes that the applied forces are acting at the center of gravity of the body.

Ports

Input

F_{xyz}(N) — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt (kg/s) — Rate of change of mass
scalar

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

V_{re} — Relative velocity
three-element vector

One or more relative velocities, specified as a three-element vector, at which the mass is accreted to or ablated from the body in body-fixed axes.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

μ γ χ (rad) — Wind rotation angles
three-element vector

Wind rotation angles [bank, flight path, heading], returned as three-element vector, in radians.

Data Types: `double`

DCM_{we} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to wind-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_w — Velocity in wind-fixed frame
three-element vector

Velocity in wind-fixed frame, returned as a three-element vector.

Data Types: `double`

α β (rad) — Angle of attack and sideslip angle
two-element vector

Angle of attack and sideslip angle, returned as a two-element vector, in radians.

Data Types: `double`

dα/dt dβ/dt — Rate of change of angle of attack and rate of change of sideslip angle
two-element vector

Rate of change of angle of attack and rate of change of sideslip angle, returned as a two-element vector, in radians per second.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations in body-fixed axes
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Simple Variable` (default) | `Fixed` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)

Mass Type	Description	Default For
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the equations of motion in “Algorithms” on page 5-843.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Quaternion` (default) | `Wind Angles`

Equations of motion representation, specified according to the following table.

Representation	Description
`Quaternion`	Use quaternions within equations of motion.
`Wind Angles`	Use wind angles within equations of motion.

The `Quaternion` selection conforms to the equations of motion in “Algorithms” on page 5-843.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Wind Angles` | `Quaternion`
Default: `'Quaternion'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial airspeed, angle of attack, and sideslip angle [V,alpha,beta] — Initial airspeed, angle of attack, and sideslip angle

`[0 0 0]` (default) | three-element vector

Initial airspeed, angle of attack, and sideslip angle, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial wind orientation [bank angle,flight path angle,heading angle] — Initial wind orientation

`[0 0 0]` (default) | three-element vector

Initial wind angles [bank, flight path, and heading], specified as a three-element vector in radians.

Programmatic Use
Block Parameter: `wind_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Empty mass — Empty mass

`0.5` (default) | scalar

Empty mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: mass_e
Type: character vector
Values: double scalar
Default: '0.5'

Full mass — Full mass of body

2.0 (default) | scalar

Full mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: mass_f
Type: character vector
Values: double scalar
Default: '2.0'

Empty inertia matrix in body axis — Inertia tensor matrix for empty inertia

eye(3) (default) | 3-by-3 matrix

Inertia tensor matrix for the empty inertia of the body, specified as 3-by-3 matrix, in body-fixed axes.

Programmatic Use
Block Parameter: inertia_e
Type: character vector
Values: 'eye(3)' | 3-by-3 matrix
Default: 'eye(3)'

Full inertia matrix in body axis — Inertia tensor matrix for full inertia

2*eye(3) (default) | 3-by-3 matrix

Inertia tensor matrix for the full inertia of the body, specified as a 3-by-3 matrix, in body-fixed axes.

Programmatic Use
Block Parameter: inertia_f
Type: character vector
Values: '2*eye(3)' | 3-by-3 matrix
Default: '2*eye(3)'

Include mass flow relative velocity — Mass flow relative velocity port

off (default) | on

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: vre_flag
Type: character vector
Values: off | on
Default: off

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use
Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state names, specified as a character vector.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

`''` (default) | character vector

Incidence angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector
Values: `''`
Default: `''`

Sideslip angle e.g., 'beta' — Sideslip angle state name

`''` (default) | character vector

Sideslip angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `beta_statename`
Type: character vector
Values: `''`
Default: `''`

Quaternion vector: e.g., {'qr', 'qi', 'qj', 'qk'} — Quaternion vector state name

`''` (default) | comma-separated list surrounded by braces

Quaternion vector state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `quat_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Mass: e.g., 'mass' — Mass state name

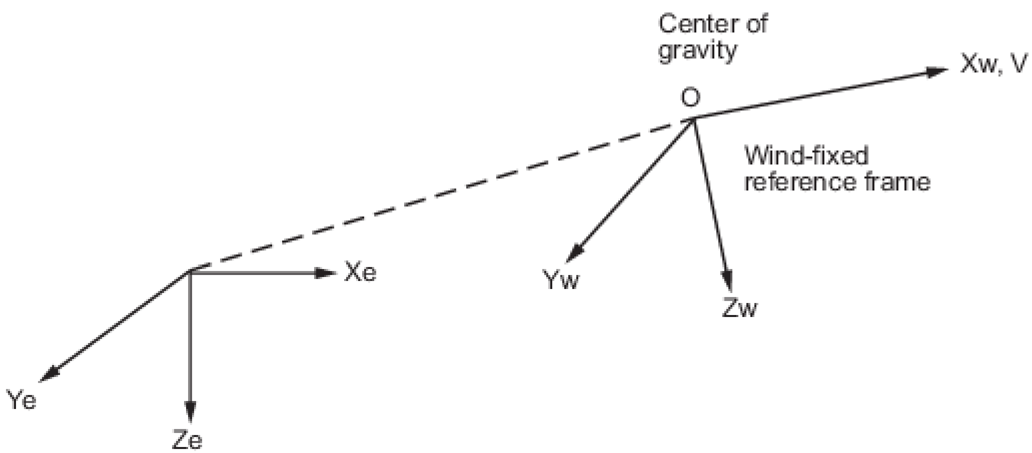
'' (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The origin of the wind-fixed coordinate frame is the center of gravity of the body, and the body is assumed to be rigid, an assumption that eliminates the need to consider the forces acting between individual elements of mass. The flat Earth reference frame is considered inertial, an excellent approximation that allows the forces due to the Earth's motion relative to the “fixed stars” to be neglected.



The translational motion of the wind-fixed coordinate frame is given below, where the applied forces $[F_x\ F_y\ F_z]^T$ are in the wind-fixed frame. Vre_w is the relative velocity in the wind axes at which the mass flow ($\dot{m}$) is ejected or added to the body.

$$\bar{F}_w = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m(\dot{\bar{V}}_w + \bar{\omega}_w \times \bar{V}_w) + \dot{m}\bar{V}re_w$$

$$\bar{V}_w = \begin{bmatrix} V \\ 0 \\ 0 \end{bmatrix}, \bar{\omega}_w = \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = DMC_{wb}\begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}, \bar{w}_b\begin{bmatrix} p_b \\ q_b \\ r_b \end{bmatrix}$$

The rotational dynamics of the body-fixed frame are given below, where the applied moments are $[L\ M\ N]^T$, and the inertia tensor I is with respect to the origin O. Inertia tensor I is much easier to define in body-fixed frame.

$$\overline{M}_b = \begin{bmatrix} L \\ M \\ N \end{bmatrix} = I\dot{\overline{\omega}}_b + \overline{\omega}_b \times (I\overline{\omega}_b) + \dot{I}\,\overline{\omega}_b$$

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

The inertia tensor is determined using a table lookup which linearly interpolates between I_{full} and I_{empty} based on mass (m). While the rate of change of the inertia tensor is estimated by the following equation.

$$\dot{I} = \frac{I_{full} - I_{empty}}{m_{full} - m_{empty}}\dot{m}$$

The integration of the rate of change of the quaternion vector is given below.

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -1/2\begin{bmatrix} 0 & p & q & r \\ -p & 0 & -r & q \\ -q & r & 0 & -p \\ -r & -q & p & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H., *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

C/C++ Code Generation

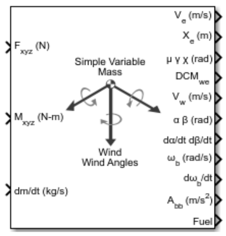
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Simple Variable Mass 6DOF Wind (Wind Angles)

Implement wind angle representation of six-degrees-of-freedom equations of motion of simple variable mass



Libraries:
Aerospace Blockset / Equations of Motion / 6DOF

Description

The Simple Variable Mass 6DOF Wind (Wind Angles) block implements a wind angle representation of six-degrees-of-freedom equations of motion of simple variable mass. For more information of the relationship between the wind angles, see Algorithms. For a description of the coordinate system employed and the translational dynamics, see the block description for the Simple Variable Mass 6DOF (Quaternion) block.

Limitations

The block assumes that the applied forces are acting at the center of gravity of the body.

Ports

Input

F_{xyz}(N) — Applied forces
three-element vector

Applied forces, specified as a three-element vector.

Data Types: `double`

M_{xyz}(N-m) — Applied moments
three-element vector

Applied moments, specified as a three-element vector.

Data Types: `double`

dm/dt (kg/s) — Rate of change of mass
scalar

One or more rates of change of mass (positive if accreted, negative if ablated), specified as a scalar.

Data Types: `double`

V_{re} — Relative velocity
three-element vector

One or more relative velocities, specified as a three-element vector, at which the mass is accreted to or ablated from the body in body-fixed axes.

Dependencies

To enable this port, select **Include mass flow relative velocity**.

Data Types: `double`

Output

V_e — Velocity in flat Earth reference frame
three-element vector

Velocity in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

X_e — Position in flat Earth reference frame
three-element vector

Position in the flat Earth reference frame, returned as a three-element vector.

Data Types: `double`

μ γ χ (rad) — Wind rotation angles
three-element vector

Wind rotation angles [bank, flight path, heading], returned as three-element vector, in radians.

Data Types: `double`

DCM_{we} — Coordinate transformation
3-by-3 matrix

Coordinate transformation from flat Earth axes to wind-fixed axes, returned as a 3-by-3 matrix.

Data Types: `double`

V_w — Velocity in wind-fixed frame
three-element vector

Velocity in wind-fixed frame, returned as a three-element vector.

Data Types: `double`

α β (rad) — Angle of attack and sideslip angle
two-element vector

Angle of attack and sideslip angle, returned as a two-element vector, in radians.

Data Types: `double`

dα/dt dβ/dt — Rate of change of angle of attack and rate of change of sideslip angle
two-element vector

Rate of change of angle of attack and rate of change of sideslip angle, returned as a two-element vector, in radians per second.

Data Types: `double`

ω_b (rad/s) — Angular rates in body-fixed axes
three-element vector

Angular rates in body-fixed axes, returned as a three-element vector.

Data Types: `double`

$d\omega_b/dt$ — Angular accelerations in body-fixed axes
three-element vector

Angular accelerations in body-fixed axes, returned as a three-element vector, in radians per second squared.

Data Types: `double`

A_{bb} — Accelerations in body-fixed axes
three-element vector

Accelerations in body-fixed axes with respect to body frame, returned as a three-element vector.

Data Types: `double`

Fuel — Fuel tank status
scalar

Fuel tank status, returned as:

- `1` — Tank is full.
- `0` — Tank is neither full nor empty.
- `-1` — Tank is empty.

Data Types: `double`

A_{be} — Accelerations with respect to inertial frame
three-element vector

Accelerations in body-fixed axes with respect to inertial frame (flat Earth), returned as a three-element vector. You typically connect this signal to the accelerometer.

Dependencies

This port appears only when the **Include inertial acceleration** check box is selected.

Data Types: `double`

Parameters

Main

Units — Input and output units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Input and output units, specified as `Metric (MKS)`, `English (Velocity in ft/s)`, or `English (Velocity in kts)`.

Units	Forces	Moment	Acceleration	Velocity	Position	Mass	Inertia
`Metric (MKS)`	Newton	Newton-meter	Meters per second squared	Meters per second	Meters	Kilogram	Kilogram meter squared
`English (Velocity in ft/s)`	Pound	Foot-pound	Feet per second squared	Feet per second	Feet	Slug	Slug foot squared
`English (Velocity in kts)`	Pound	Foot-pound	Feet per second squared	Knots	Feet	Slug	Slug foot squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `Metric (MKS)` | `English (Velocity in ft/s)` | `English (Velocity in kts)`
Default: `Metric (MKS)`

Mass Type — Mass type

`Simple Variable` (default) | `Fixed` | `Custom Variable`

Mass type, specified according to the following table.

Mass Type	Description	Default For
`Fixed`	Mass is constant throughout the simulation.	• 6DOF (Euler Angles) • 6DOF (Quaternion) • 6DOF Wind (Wind Angles) • 6DOF Wind (Quaternion) • 6DOF ECEF (Quaternion)
`Simple Variable`	Mass and inertia vary linearly as a function of mass rate.	• Simple Variable Mass 6DOF (Euler Angles) • Simple Variable Mass 6DOF (Quaternion) • Simple Variable Mass 6DOF Wind (Wind Angles) • Simple Variable Mass 6DOF Wind (Quaternion) • Simple Variable Mass 6DOF ECEF (Quaternion)

Mass Type	Description	Default For
`Custom Variable`	Mass and inertia variations are customizable.	• Custom Variable Mass 6DOF (Euler Angles) • Custom Variable Mass 6DOF (Quaternion) • Custom Variable Mass 6DOF Wind (Wind Angles) • Custom Variable Mass 6DOF Wind (Quaternion) • Custom Variable Mass 6DOF ECEF (Quaternion)

The `Simple Variable` selection conforms to the equations of motion in "Algorithms" on page 5-854.

Programmatic Use
Block Parameter: `mtype`
Type: character vector
Values: `Fixed` | `Simple Variable` | `Custom Variable`
Default: `Simple Variable`

Representation — Equations of motion representation

`Wind Angles` (default) | `Quaternion`

Equations of motion representation, specified according to the following table.

Representation	Description
`Wind Angles`	Use Wind angles within equations of motion.
`Quaternion`	Use quaternions within equations of motion.

The `Wind Angles` selection conforms to the equations of motion in "Algorithms" on page 5-854.

Programmatic Use
Block Parameter: `rep`
Type: character vector
Values: `Wind Angles` | `Quaternion`
Default: `'Wind Angles'`

Initial position in inertial axes [Xe,Ye,Ze] — Position in inertial axes

`[0 0 0]` (default) | three-element vector

Initial location of the body in the flat Earth reference frame, specified as a three-element vector.

Programmatic Use
Block Parameter: `xme_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial airspeed, angle of attack, and sideslip angle [V,alpha,beta] — Initial airspeed, angle of attack, and sideslip angle

`[0 0 0]` (default) | three-element vector

Initial airspeed, angle of attack, and sideslip angle, specified as a three-element vector.

Programmatic Use
Block Parameter: `Vm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial wind orientation [bank angle,flight path angle,heading angle] — Initial wind orientation

`[0 0 0]` (default) | three-element vector

Initial wind angles [bank, flight path, and heading], specified as a three-element vector in radians.

Programmatic Use
Block Parameter: `wind_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial body rotation rates [p,q,r] — Initial body rotation

`[0 0 0]` (default) | three-element vector

Initial body-fixed angular rates with respect to the NED frame, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `pm_0`
Type: character vector
Values: `'[0 0 0]'` | three-element vector
Default: `'[0 0 0]'`

Initial mass — Initial mass

`1.0` (default) | scalar

Initial mass of the rigid body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_0`
Type: character vector
Values: `'1.0'` | double scalar
Default: `'1.0'`

Empty mass — Empty mass

`0.5` (default) | scalar

Empty mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_e`
Type: character vector
Values: double scalar
Default: `'0.5'`

Full mass — Full mass of body

`2.0` (default) | scalar

Full mass of the body, specified as a double scalar.

Programmatic Use
Block Parameter: `mass_f`
Type: character vector
Values: double scalar
Default: `'2.0'`

Empty inertia matrix in body axis — Inertia tensor matrix for empty inertia

`eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the empty inertia of the body, specified as 3-by-3 matrix, in body-fixed axes.

Programmatic Use
Block Parameter: `inertia_e`
Type: character vector
Values: `'eye(3)'` | 3-by-3 matrix
Default: `'eye(3)'`

Full inertia matrix in body axis — Inertia tensor matrix for full inertia

`2*eye(3)` (default) | 3-by-3 matrix

Inertia tensor matrix for the full inertia of the body, specified as a 3-by-3 matrix, in body-fixed axes.

Programmatic Use
Block Parameter: `inertia_f`
Type: character vector
Values: `'2*eye(3)'` | 3-by-3 matrix
Default: `'2*eye(3)'`

Include mass flow relative velocity — Mass flow relative velocity port

`off` (default) | `on`

Select this check box to add a mass flow relative velocity port. This is the relative velocity at which the mass is accreted or ablated.

Programmatic Use
Block Parameter: `vre_flag`
Type: character vector
Values: `off` | `on`
Default: `off`

Include inertial acceleration — Include inertial acceleration port

`off` (default) | `on`

Select this check box to add an inertial acceleration port.

Dependencies

To enable the $\mathbf{A_{b\ ff}}$ port, select this parameter.

Programmatic Use

Block Parameter: `abi_flag`
Type: character vector
Values: `'off'` | `'on'`
Default: `off`

State Attributes

Assign a unique name to each state. You can use state names instead of block paths during linearization.

- To assign a name to a single state, enter a unique name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-separated list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- If a parameter is empty (`' '`), no name is assigned.
- The state names apply only to the selected block with the name parameter.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

  For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Position: e.g., {'Xe', 'Ye', 'Ze'} — Position state name

`''` (default) | comma-separated list surrounded by braces

Position state names, specified as a comma-separated list surrounded by braces.

Programmatic Use

Block Parameter: `xme_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Velocity: e.g., 'V' — Velocity state name

`''` (default) | character vector

Velocity state names, specified as a character vector.

Programmatic Use
Block Parameter: `Vm_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Incidence angle e.g., 'alpha' — Incidence angle state name

`''` (default) | character vector

Incidence angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `alpha_statename`
Type: character vector
Values: `''`
Default: `''`

Sideslip angle e.g., 'beta' — Sideslip angle state name

`''` (default) | character vector

Sideslip angle state name, specified as a character vector.

Programmatic Use
Block Parameter: `beta_statename`
Type: character vector
Values: `''`
Default: `''`

Wind orientation e.g., {'mu', 'gamma', 'chi'} — Wind orientation state names

`''` (default) | comma-separated list surrounded by braces

Wind orientation state names, specified as a comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `wind_statename`
Type: character vector
Values: `''`
Default: `''`

Body rotation rates: e.g., {'p', 'q', 'r'} — Body rotation state names

`''` (default) | comma-separated list surrounded by braces

Body rotation rate state names, specified comma-separated list surrounded by braces.

Programmatic Use
Block Parameter: `pm_statename`
Type: character vector
Values: `''` | comma-separated list surrounded by braces
Default: `''`

Mass: e.g., 'mass' — Mass state name

'' (default) | character vector

Mass state name, specified as a character vector.

Programmatic Use
Block Parameter: `mass_statename`
Type: character vector
Values: `''` | character vector
Default: `''`

Algorithms

The relationship between the wind angles, $[\mu\gamma\chi]^{\mathrm{T}}$, can be determined by resolving the wind rates into the wind-fixed coordinate frame.

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} \dot{\mu} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\gamma} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} \cos\gamma & 0 & -\sin\gamma \\ 0 & 1 & 0 \\ \sin\gamma & 0 & \cos\gamma \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\chi} \end{bmatrix} \equiv J^{-1} \begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix}$$

Inverting J then gives the required relationship to determine the wind rate vector.

$$\begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix} = J \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} 1 & (\sin\mu\tan\gamma) & (\cos\mu\tan\gamma) \\ 0 & \cos\mu & -\sin\mu \\ 0 & \dfrac{\sin\mu}{\cos\gamma} & \dfrac{\cos\mu}{\cos\gamma} \end{bmatrix} \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix}$$

The body-fixed angular rates are related to the wind-fixed angular rate by the following equation.

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = DMC_{wb} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Using this relationship in the wind rate vector equations, gives the relationship between the wind rate vector and the body-fixed angular rates.

$$\begin{bmatrix} \dot{\mu} \\ \dot{\gamma} \\ \dot{\chi} \end{bmatrix} = J \begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} 1 & (\sin\mu\tan\gamma) & (\cos\mu\tan\gamma) \\ 0 & \cos\mu & -\sin\mu \\ 0 & \dfrac{\sin\mu}{\cos\gamma} & \dfrac{\cos\mu}{\cos\gamma} \end{bmatrix} DMC_{wb} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Version History

Introduced in R2006a

References

[1] Stevens, Brian, and Frank Lewis. *Aircraft Control and Simulation*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Zipfel, Peter H. *Modeling and Simulation of Aerospace Vehicle Dynamics*. 2nd ed. Reston, VA: AIAA Education Series, 2007.

Extended Capabilities

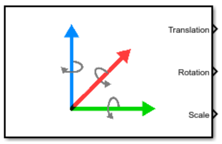
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

6DOF (Euler Angles) | 6DOF (Quaternion) | 6DOF ECEF (Quaternion) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 6DOF (Euler Angles) | Custom Variable Mass 6DOF (Quaternion) | Custom Variable Mass 6DOF ECEF (Quaternion) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 6DOF ECEF (Quaternion) | Simple Variable Mass 6DOF (Euler Angles) | Simple Variable Mass 6DOF (Quaternion) | Simple Variable Mass 6DOF Wind (Quaternion)

Simulation 3D Actor Transform Get

Get actor translation, rotation, scale



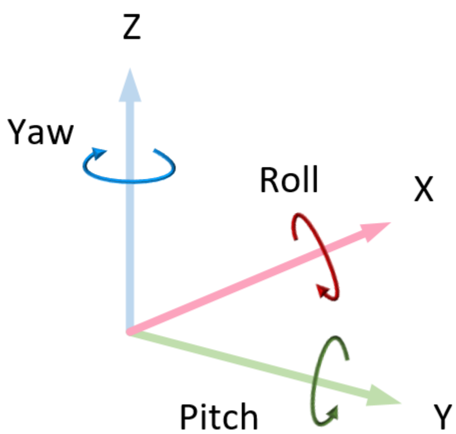
Libraries:
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core
Aerospace Blockset / Animation / Simulation 3D
Simulink 3D Animation / Simulation 3D / Utilities

Description

Note Simulating models with the Simulation 3D Actor Transform Get block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Actor Transform Get to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Actor Transform Get block provides the actor translation, rotation, and scale for the Simulink simulation environment.

The block uses a vehicle-fixed coordinate system that is initially aligned with the inertial world coordinate system.



Axis	Description
X	Forward direction of the vehicle Roll — Right-handed rotation about *X*-axis
Y	Extends to the right of the vehicle, initially parallel to the ground plane Pitch — Right-handed rotation about *Y*-axis

Axis	Description
Z	Extends upwards Yaw — Left-handed rotation about *Z*-axis

Actors are scene objects that support 3D translation, rotation, and scale. Parts are actor components. Components do not exist by themselves; they are associated with an actor.

Tip Verify that the Simulation 3D Scene Configuration block executes before the Simulation 3D Actor Transform Get block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Actor Transform Get block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Actor Transform Get — `1`

For more information about execution order, see "Control and Display Execution Order".

Ports

Output

Translation — Actor translation
`array`

Actor translation, in m. Array dimensions are number of parts per actor-by-3.

- `Translation(1,1)`, `Translation(1,2)`, and `Translation(1,3)` — Vehicle displacement along world *X*-, *Y*, and *Z*- axes, respectively.
- `Translation(...,1)`, `Translation(...,2)`, and `Translation(...,3)` — Actor displacement relative to vehicle, in vehicle-fixed coordinate system initially aligned with world *X*-, *Y*, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The `Translation` signal:

- Dimensions are `[5x3]`.
- Contains translation information according to the axle and wheel locations, relative to vehicle.

$$Translation = \begin{bmatrix} X_v & Y_v & Z_v \\ X_{FL} & Y_{FL} & Z_{FL} \\ X_{FR} & Y_{FR} & Z_{FR} \\ X_{RL} & Y_{RL} & Z_{RL} \\ X_{RR} & Y_{RR} & Z_{RR} \end{bmatrix}$$

Translation	Array Element
Vehicle, X_v	`Translation(1,1)`
Vehicle, Y_v	`Translation(1,2)`
Vehicle, Z_v	`Translation(1,3)`

Translation	Array Element
Front left wheel, X_{FL}	`Translation(2,1)`
Front left wheel, Y_{FL}	`Translation(2,2)`
Front left wheel, Z_{FL}	`Translation(2,3)`
Front right wheel, X_{FR}	`Translation(3,1)`
Front right wheel, Y_{FR}	`Translation(3,2)`
Front right wheel, Z_{FR}	`Translation(3,3)`
Rear left wheel, X_{RL}	`Translation(4,1)`
Rear left wheel, Y_{RL}	`Translation(4,2)`
Rear left wheel, Z_{RL}	`Translation(4,3)`
Rear right wheel, X_{RR}	`Translation(5,1)`
Rear right wheel, Y_{RR}	`Translation(5,2)`
Rear right wheel, Z_{RR}	`Translation(5,3)`

Rotation — Actor rotation
`array`

Actor rotation across a [-pi/2, pi/2] range, in rad. Array dimensions are number of parts per actor-by-3.

- `Rotation(1,1)`, `Rotation(1,2)`, and `Rotation(1,3)` — Vehicle rotation about vehicle-fixed pitch, roll, and yaw *Y*-, *Z*-, and *X*- axes, respectively.
- `Rotation(...,1)`, `Rotation(...,2)`, and `Rotation(...,3)` — Actor rotation about vehicle-fixed pitch, roll, and yaw *Y*-, *X*-, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The `Rotation` signal:

- Dimensions are `[5x3]`.
- Contains rotation information according to the axle and wheel locations.

$$Rotation = \begin{bmatrix} Pitch_v & Roll_v & Yaw_v \\ Pitch_{FL} & Roll_{FL} & Yaw_{FL} \\ Pitch_{FR} & Roll_{FR} & Yaw_{FR} \\ Pitch_{RL} & Roll_{RL} & Yaw_{RL} \\ Pitch_{RR} & Roll_{RR} & Yaw_{RR} \end{bmatrix}$$

Rotation	Array Element
Vehicle, $Pitch_v$	`Rotation(1,1)`
Vehicle, $Roll_v$	`Rotation(1,2)`
Vehicle, Yaw_v	`Rotation(1,3)`
Front left wheel, $Pitch_{FL}$	`Rotation(2,1)`
Front left wheel, $Roll_{FL}$	`Rotation(2,2)`
Front left wheel, Yaw_{FL}	`Rotation(2,3)`

Rotation	**Array Element**
Front right wheel, $Pitch_{FR}$	`Rotation(3,1)`
Front right wheel, $Roll_{FR}$	`Rotation(3,2)`
Front right wheel, Yaw_{FR}	`Rotation(3,3)`
Rear left wheel, $Pitch_{RL}$	`Rotation(4,1)`
Rear left wheel, $Roll_{RL}$	`Rotation(4,2)`
Rear left wheel, Yaw_{RL}	`Rotation(4,3)`
Rear right wheel, $Pitch_{RR}$	`Rotation(5,1)`
Rear right wheel, $Roll_{RR}$	`Rotation(5,2)`
Rear right wheel, Yaw_{RR}	`Rotation(5,3)`

Scale — Actor scale
`array`

Actor scale. Array dimensions are number of parts per actor-by-3.

- `Scale(1,1)`, `Scale(1,2)`, and `Scale(1,3)` — Vehicle scale along world *X*-, *Y*-, and *Z*- axes, respectively.
- `Scale(...,1)`, `Scale(...,2)`, and `Scale(...,3)` — Actor scale along world *X*-, *Y*-, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The `Scale` signal:

- Dimensions are `[5x3]`.
- Contains scale information according to the axle and wheel locations.

$$Scale = \begin{bmatrix} X_{V_{scale}} & Y_{V_{scale}} & Z_{V_{scale}} \\ X_{FL_{scale}} & Y_{FL_{scale}} & Z_{FL_{scale}} \\ X_{FR_{scale}} & Y_{FR_{scale}} & Z_{FR_{scale}} \\ X_{RL_{scale}} & Y_{RL_{scale}} & Z_{RL_{scale}} \\ X_{RR_{scale}} & Y_{RR_{scale}} & Z_{RR_{scale}} \end{bmatrix}$$

Scale	**Array Element**
Vehicle, $X_{v_{scale}}$	`Scale(1,1)`
Vehicle, $Y_{v_{scale}}$	`Scale(1,2)`
Vehicle, $Z_{v_{scale}}$	`Scale(1,3)`
Front left wheel, $X_{FL_{scale}}$	`Scale(2,1)`
Front left wheel, $Y_{FL_{scale}}$	`Scale(2,2)`
Front left wheel, $Z_{FL_{scale}}$	`Scale(2,3)`
Front right wheel, $X_{FR_{scale}}$	`Scale(3,1)`
Front right wheel, $Y_{FR_{scale}}$	`Scale(3,2)`
Front right wheel, $Z_{FR_{scale}}$	`Scale(3,3)`

Scale	Array Element
Rear left wheel, $X_{RL_{scale}}$	`Scale(4,1)`
Rear left wheel, $Y_{RL_{scale}}$	`Scale(4,2)`
Rear left wheel, $Z_{RL_{scale}}$	`Scale(4,3)`
Rear right wheel, $X_{RR_{scale}}$	`Scale(5,1)`
Rear right wheel, $Y_{RR_{scale}}$	`Scale(5,2)`
Rear right wheel, $Z_{RR_{scale}}$	`Scale(5,3)`

Parameters

Tag for actor in 3D scene, ActorTag — Name

`SimulinkActor1` (default) | character vector

Actor name.

Actors are scene objects that support 3D translation, rotation, and scale. Parts are actor components. Components do not exist by themselves; they are associated with an actor.

The block does not support multiple instances of the same actor tag. To refer to the same scene actor when you use the 3D block pairs (e.g. Simulation 3D Actor Transform Get and Simulation 3D Actor Transform Set), specify the same **Tag for actor in 3D scene, ActorTag** parameter.

Number of parts per actor to get, NumberOfParts — Name

`1` (default) | `scalar`

Number of parts per actor. Actors are scene objects that support 3D translation, rotation, and scale. Parts are actor components. Components do not exist by themselves; they are associated with an actor. Typically, a vehicle actor with a body and four wheels has 5 parts.

The block does not support multiple instances of the same actor tag. To refer to the same scene actor when you use the 3D block pairs (e.g. Simulation 3D Actor Transform Get and Simulation 3D Actor Transform Set), specify the same **Tag for actor in 3D scene, ActorTag** parameter.

Sample time — Sample time

`-1` (default) | `scalar`

Sample time, T_s. The graphics frame rate is the inverse of the sample time.

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

Behavior change in future release

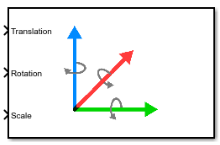
Simulating models with the Simulation 3D Actor Transform Get block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Actor Transform Get to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

See Also

Simulation 3D Scene Configuration

Simulation 3D Actor Transform Set

Set actor translation, rotation, scale



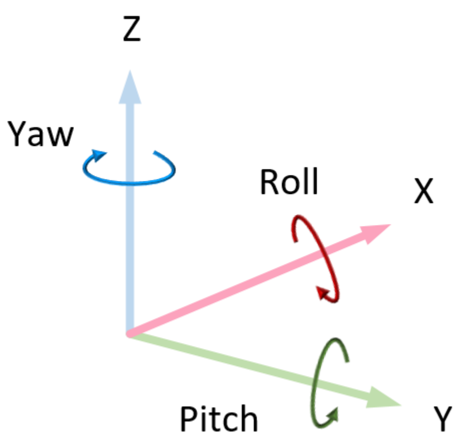
Libraries:
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core
Aerospace Blockset / Animation / Simulation 3D
Simulink 3D Animation / Simulation 3D / Utilities

Description

Note Simulating models with the Simulation 3D Actor Transform Set block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Actor Transform Set to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Actor Transform Set block sets the actor translation, rotation, and scale in the 3D visualization environment.

The block uses a vehicle-fixed coordinate system that is initially aligned with the inertial world coordinate system.



Axis	Description
X	Forward direction of the vehicle Roll — Right-handed rotation about *X*-axis
Y	Extends to the right of the vehicle, initially parallel to the ground plane Pitch — Right-handed rotation about *Y*-axis

Axis	Description
Z	Extends upwards Yaw — Left-handed rotation about Z-axis

Actors are scene objects that support 3D translation, rotation, and scale. Parts are actor components. Components do not exist by themselves; they are associated with an actor.

Tip Verify that the Simulation 3D Actor Transform Set block executes before the Simulation 3D Scene Configuration block. That way, Simulation 3D Actor Transform Set prepares the signal data before the Unreal Engine 3D visualization environment receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Actor Transform Set — `-1`

For more information about execution order, see "Control and Display Execution Order".

Ports

Input

Translation — Actor translation
`array`

Actor translation, in m. Array dimensions are number of parts per actor-by-`3`.

- `Translation(1,1)`, `Translation(1,2)`, and `Translation(1,3)` — Vehicle displacement along world X-, Y, and Z- axes, respectively.
- `Translation(...,1)`, `Translation(...,2)`, and `Translation(...,3)` — Actor displacement relative to vehicle, in vehicle-fixed coordinate system initially aligned with world X-, Y, and Z- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The `Translation` signal:

- Dimensions are `[5x3]`.
- Contains translation information according to the axle and wheel locations, relative to vehicle.

$$Translation = \begin{bmatrix} X_v & Y_v & Z_v \\ X_{FL} & Y_{FL} & Z_{FL} \\ X_{FR} & Y_{FR} & Z_{FR} \\ X_{RL} & Y_{RL} & Z_{RL} \\ X_{RR} & Y_{RR} & Z_{RR} \end{bmatrix}$$

Translation	Array Element
Vehicle, X_v	`Translation(1,1)`
Vehicle, Y_v	`Translation(1,2)`

Translation	Array Element
Vehicle, Z_v	`Translation(1,3)`
Front left wheel, X_{FL}	`Translation(2,1)`
Front left wheel, Y_{FL}	`Translation(2,2)`
Front left wheel, Z_{FL}	`Translation(2,3)`
Front right wheel, X_{FR}	`Translation(3,1)`
Front right wheel, Y_{FR}	`Translation(3,2)`
Front right wheel, Z_{FR}	`Translation(3,3)`
Rear left wheel, X_{RL}	`Translation(4,1)`
Rear left wheel, Y_{RL}	`Translation(4,2)`
Rear left wheel, Z_{RL}	`Translation(4,3)`
Rear right wheel, X_{RR}	`Translation(5,1)`
Rear right wheel, Y_{RR}	`Translation(5,2)`
Rear right wheel, Z_{RR}	`Translation(5,3)`

Rotation — Actor rotation
`array`

Actor rotation across a [-pi/2, pi/2] range, in rad. Array dimensions are number of parts per actor-by-3.

- `Rotation(1,1)`, `Rotation(1,2)`, and `Rotation(1,3)` — Vehicle rotation about vehicle-fixed pitch, roll, and yaw *Y*-, *Z*-, and *X*- axes, respectively.
- `Rotation(...,1)`, `Rotation(...,2)`, and `Rotation(...,3)` — Actor rotation about vehicle-fixed pitch, roll, and yaw *Y*-, *X*-, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The `Rotation` signal:

- Dimensions are `[5x3]`.
- Contains rotation information according to the axle and wheel locations.

$$Rotation = \begin{bmatrix} Pitch_v & Roll_v & Yaw_v \\ Pitch_{FL} & Roll_{FL} & Yaw_{FL} \\ Pitch_{FR} & Roll_{FR} & Yaw_{FR} \\ Pitch_{RL} & Roll_{RL} & Yaw_{RL} \\ Pitch_{RR} & Roll_{RR} & Yaw_{RR} \end{bmatrix}$$

Rotation	Array Element
Vehicle, $Pitch_v$	`Rotation(1,1)`
Vehicle, $Roll_v$	`Rotation(1,2)`
Vehicle, Yaw_v	`Rotation(1,3)`
Front left wheel, $Pitch_{FL}$	`Rotation(2,1)`
Front left wheel, $Roll_{FL}$	`Rotation(2,2)`

Rotation	Array Element
Front left wheel, Yaw_{FL}	`Rotation(2,3)`
Front right wheel, $Pitch_{FR}$	`Rotation(3,1)`
Front right wheel, $Roll_{FR}$	`Rotation(3,2)`
Front right wheel, Yaw_{FR}	`Rotation(3,3)`
Rear left wheel, $Pitch_{RL}$	`Rotation(4,1)`
Rear left wheel, $Roll_{RL}$	`Rotation(4,2)`
Rear left wheel, Yaw_{RL}	`Rotation(4,3)`
Rear right wheel, $Pitch_{RR}$	`Rotation(5,1)`
Rear right wheel, $Roll_{RR}$	`Rotation(5,2)`
Rear right wheel, Yaw_{RR}	`Rotation(5,3)`

Scale — Actor scale
`array`

Actor scale. Array dimensions are number of parts per actor-by-3.

- `Scale(1,1)`, `Scale(1,2)`, and `Scale(1,3)` — Vehicle scale along world *X*-, *Y*-, and *Z*- axes, respectively.
- `Scale(...,1)`, `Scale(...,2)`, and `Scale(...,3)` — Actor scale along world *X*-, *Y*-, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The `Scale` signal:

- Dimensions are `[5x3]`.
- Contains scale information according to the axle and wheel locations.

$$Scale = \begin{bmatrix} X_{V_{scale}} & Y_{V_{scale}} & Z_{V_{scale}} \\ X_{FL_{scale}} & Y_{FL_{scale}} & Z_{FL_{scale}} \\ X_{FR_{scale}} & Y_{FR_{scale}} & Z_{FR_{scale}} \\ X_{RL_{scale}} & Y_{RL_{scale}} & Z_{RL_{scale}} \\ X_{RR_{scale}} & Y_{RR_{scale}} & Z_{RR_{scale}} \end{bmatrix}$$

Scale	Array Element
Vehicle, $X_{v_{scale}}$	`Scale(1,1)`
Vehicle, $Y_{v_{scale}}$	`Scale(1,2)`
Vehicle, $Z_{v_{scale}}$	`Scale(1,3)`
Front left wheel, $X_{FL_{scale}}$	`Scale(2,1)`
Front left wheel, $Y_{FL_{scale}}$	`Scale(2,2)`
Front left wheel, $Z_{FL_{scale}}$	`Scale(2,3)`
Front right wheel, $X_{FR_{scale}}$	`Scale(3,1)`
Front right wheel, $Y_{FR_{scale}}$	`Scale(3,2)`

Scale	Array Element
Front right wheel, $Z_{FR_{scale}}$	`Scale(3,3)`
Rear left wheel, $X_{RL_{scale}}$	`Scale(4,1)`
Rear left wheel, $Y_{RL_{scale}}$	`Scale(4,2)`
Rear left wheel, $Z_{RL_{scale}}$	`Scale(4,3)`
Rear right wheel, $X_{RR_{scale}}$	`Scale(5,1)`
Rear right wheel, $Y_{RR_{scale}}$	`Scale(5,2)`
Rear right wheel, $Z_{RR_{scale}}$	`Scale(5,3)`

Parameters

Actor Setup

Tag for actor in 3D scene, ActorTag — Name

`SimulinkActor1` (default) | character vector

Actor name.

Actors are scene objects that support 3D translation, rotation, and scale. Parts are actor components. Components do not exist by themselves; they are associated with an actor.

The block does not support multiple instances of the same actor tag. To refer to the same scene actor when you use the 3D block pairs (e.g. Simulation 3D Actor Transform Get and Simulation 3D Actor Transform Set), specify the same **Tag for actor in 3D scene, ActorTag** parameter.

Number of parts per actor to set, NumberOfParts — Name

`1` (default) | `scalar`

Number of parts per actor. Actors are scene objects that support 3D translation, rotation, and scale. Parts are actor components. Components do not exist by themselves; they are associated with an actor. Typically, a vehicle actor with a body and four wheels has 5 parts.

The block does not support multiple instances of the same actor tag. To refer to the same scene actor when you use the 3D block pairs (e.g. Simulation 3D Actor Transform Get and Simulation 3D Actor Transform Set), specify the same **Tag for actor in 3D scene, ActorTag** parameter.

Initial Values

Initial array values to translate actor per part, Translation — Actor initial position

`[0 0 0]` (default) | `array`

Actor initial position, along world *X*-, *Y*-, and *Z*- axes, in m.

Array dimensions are number of parts per actor-by-3.

- `Translation(1,1)`, `Translation(1,2)`, and `Translation(1,3)` — Vehicle displacement along world *X*-, *Y*, and *Z*- axes, respectively.

- `Translation(...,1)`, `Translation(...,2)`, and `Translation(...,3)` — Actor displacement relative to vehicle, in vehicle-fixed coordinate system initially aligned with world *X*-, *Y*, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The parameter:

- Dimensions are `[5x3]`.
- Contains translation information according to the axle and wheel locations, relative to vehicle.

$$Translation = \begin{bmatrix} X_v & Y_v & Z_v \\ X_{FL} & Y_{FL} & Z_{FL} \\ X_{FR} & Y_{FR} & Z_{FR} \\ X_{RL} & Y_{RL} & Z_{RL} \\ X_{RR} & Y_{RR} & Z_{RR} \end{bmatrix}$$

Translation	Array Element
Vehicle, X_v	`Translation(1,1)`
Vehicle, Y_v	`Translation(1,2)`
Vehicle, Z_v	`Translation(1,3)`
Front left wheel, X_{FL}	`Translation(2,1)`
Front left wheel, Y_{FL}	`Translation(2,2)`
Front left wheel, Z_{FL}	`Translation(2,3)`
Front right wheel, X_{FR}	`Translation(3,1)`
Front right wheel, Y_{FR}	`Translation(3,2)`
Front right wheel, Z_{FR}	`Translation(3,3)`
Rear left wheel, X_{RL}	`Translation(4,1)`
Rear left wheel, Y_{RL}	`Translation(4,2)`
Rear left wheel, Z_{RL}	`Translation(4,3)`
Rear right wheel, X_{RR}	`Translation(5,1)`
Rear right wheel, Y_{RR}	`Translation(5,2)`
Rear right wheel, Z_{RR}	`Translation(5,3)`

Initial array values to rotate actor per part, Rotation — Actor initial rotation

`[0 0 0]` (default) | `array`

Actor initial rotation about world *X*-, *Y*-, and *Z*- axes across a [-pi/2, pi/2] range, in rad.

Array dimensions are number of parts per actor-by-3.

- `Rotation(1,1)`, `Rotation(1,2)`, and `Rotation(1,3)` — Vehicle rotation about vehicle-fixed pitch, roll, and yaw *Y*-, *Z*-, and *X*- axes, respectively.
- `Rotation(...,1)`, `Rotation(...,2)`, and `Rotation(...,3)` — Actor rotation about vehicle-fixed pitch, roll, and yaw *Y*-, *Z*-, and *X*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The parameter:

- Dimensions are [5x3].
- Contains rotation information according to the axle and wheel locations.

$$Rotation = \begin{bmatrix} Pitch_v & Roll_v & Yaw_v \\ Pitch_{FL} & Roll_{FL} & Yaw_{FL} \\ Pitch_{FR} & Roll_{FR} & Yaw_{FR} \\ Pitch_{RL} & Roll_{RL} & Yaw_{RL} \\ Pitch_{RR} & Roll_{RR} & Yaw_{RR} \end{bmatrix}$$

Rotation	Array Element
Vehicle, $Pitch_v$	`Rotation(1,1)`
Vehicle, $Roll_v$	`Rotation(1,2)`
Vehicle, Yaw_v	`Rotation(1,3)`
Front left wheel, $Pitch_{FL}$	`Rotation(2,1)`
Front left wheel, $Roll_{FL}$	`Rotation(2,2)`
Front left wheel, Yaw_{FL}	`Rotation(2,3)`
Front right wheel, $Pitch_{FR}$	`Rotation(3,1)`
Front right wheel, $Roll_{FR}$	`Rotation(3,2)`
Front right wheel, Yaw_{FR}	`Rotation(3,3)`
Rear left wheel, $Pitch_{RL}$	`Rotation(4,1)`
Rear left wheel, $Roll_{RL}$	`Rotation(4,2)`
Rear left wheel, Yaw_{RL}	`Rotation(4,3)`
Rear right wheel, $Pitch_{RR}$	`Rotation(5,1)`
Rear right wheel, $Roll_{RR}$	`Rotation(5,2)`
Rear right wheel, Yaw_{RR}	`Rotation(5,3)`

Initial array values to scale actor per part, Scale — Actor initial scale

`[1 1 1]` (default) | `array`

Actor initial scale.

Array dimensions are number of number of parts per actor-by-3.

- `Scale(1,1)`, `Scale(1,2)`, and `Scale(1,3)` — Vehicle scale along world *X*-, *Y*, and *Z*- axes, respectively.
- `Scale(...,1)`, `Scale(...,2)`, and `Scale(...,3)` — Actor scale along world *X*-, *Y*, and *Z*- axes, respectively.

For example, consider a vehicle actor with a vehicle body and four wheels. The parameter:

- Dimensions are [5x3].
- Contains scale information according to the axle and wheel locations.

$$Scale = \begin{bmatrix} X_{V_{scale}} & Y_{V_{scale}} & Z_{V_{scale}} \\ X_{FL_{scale}} & Y_{FL_{scale}} & Z_{FL_{scale}} \\ X_{FR_{scale}} & Y_{FR_{scale}} & Z_{FR_{scale}} \\ X_{RL_{scale}} & Y_{RL_{scale}} & Z_{RL_{scale}} \\ X_{RR_{scale}} & Y_{RR_{scale}} & Z_{RR_{scale}} \end{bmatrix}$$

Scale	Array Element	Scale Axis
Vehicle, $X_{v_{scale}}$	`Scale(1,1)`	World X-axis
Vehicle, $Y_{v_{scale}}$	`Scale(1,2)`	World Y-axis
Vehicle, $Z_{v_{scale}}$	`Scale(1,3)`	World Z-axis
Front left wheel, $X_{FL_{scale}}$	`Scale(2,1)`	World X-axis
Front left wheel, $Y_{FL_{scale}}$	`Scale(2,2)`	World Y-axis
Front left wheel, $Z_{FL_{scale}}$	`Scale(2,3)`	World Z-axis
Front right wheel, $X_{FR_{scale}}$	`Scale(3,1)`	World X-axis
Front right wheel, $Y_{FR_{scale}}$	`Scale(3,2)`	World Y-axis
Front right wheel, $Z_{FR_{scale}}$	`Scale(3,3)`	World Z-axis
Rear left wheel, $X_{RL_{scale}}$	`Scale(4,1)`	World X-axis
Rear left wheel, $Y_{RL_{scale}}$	`Scale(4,2)`	World Y-axis
Rear left wheel, $Z_{RL_{scale}}$	`Scale(4,3)`	World Z-axis
Rear right wheel, $X_{RR_{scale}}$	`Scale(5,1)`	World X-axis
Rear right wheel, $Y_{RR_{scale}}$	`Scale(5,2)`	World Y-axis
Rear right wheel, $Z_{RR_{scale}}$	`Scale(5,3)`	World Z-axis

Sample time — Sample time

`-1` (default) | `scalar`

Sample time, T_s. The graphics frame rate is the inverse of the sample time.

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

Simulating models with the Simulation 3D Actor Transform Set block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Actor Transform Set to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

See Also

Simulation 3D Scene Configuration

Simulation 3D Airliner Pack

Generate translation and rotation information for generic airliner

Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Airliner Pack block creates translation and rotation information for the Simulation 3D Aircraft block with **Type** set to `Airliner`. Use the Simulation 3D Airliner Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Aircraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Left_Engine_R — Left engine rotation
1-by-3 matrix

Left engine rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left engine rotation** parameter.

Data Types: `single` | `double`

Right_Engine_R — Right engine rotation
1-by-3 matrix

Right engine rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right engine rotation** parameter.

Data Types: `single` | `double`

Rudder_R — Rudder rotation
1-by-3 matrix

Rudder rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rudder rotation** parameter.

Data Types: `single` | `double`

Elevator_R — Elevator rotation
1-by-3 matrix

Elevator rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Elevator rotation** parameter.

Data Types: `single` | `double`

Left_Aileron_R — Left aileron rotation
1-by-3 matrix

Left aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left aileron rotation** parameter.

Data Types: `single` | `double`

Right_Aileron_R — Right aileron rotation
1-by-3 matrix

Right aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right aileron rotation** parameter.

Data Types: `single` | `double`

Flaps_R — Flaps rotation
1-by-3 matrix

Flaps rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_Strut_R — Nose wheel strut rotation
1-by-3 matrix

Nose wheel strut rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel strut rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_R — Nose wheel rotation
1-by-3 matrix

Nose wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel rotation** parameter.

Data Types: `single` | `double`

Left_Wheel_R — Left wheel rotation
1-by-3 matrix

Left wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel rotation** parameter.

Data Types: `single` | `double`

Right_Wheel_R — Right wheel rotation
1-by-3 matrix

Right wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Aircraft translation
12-by-3 array

Aircraft translation for generic airliner aircraft, returned as a 12-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the aircraft.

The translation applies to these bones of the `Airliner` type.

Bone	Index
BODY	1
LEFT_ENGINE	2
RIGHT_ENGINE	3
RUDDER	4

Bone	Index
ELEVATOR	5
LEFT_AILERON	6
RIGHT_AILERON	7
FLAPS	8
NOSE_WHEEL_STRUT	9
NOSE_WHEEL	10
LEFT_WHEEL	11
RIGHT_WHEEL	12

Rotation — Aircraft and wheel rotation
12-by-3 array

Aircraft and wheel rotation for generic airliner, returned as a 12-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the aircraft.

Parameters

Propulsion

Left engine rotation — Option to enable Left_Engine_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Engine_R** input port.

Programmatic Use
Block Parameter: `Left_Engine_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right engine rotation — Option to enable Right_Engine_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Engine_R** input port.

Programmatic Use
Block Parameter: `Right_Engine_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Controls

Rudder rotation — Option to enable Rudder_R input port

`off` (default) | `on`

Select this parameter to enable the **Rudder_R** input port.

Programmatic Use
Block Parameter: `Rudder_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Elevator rotation — Option to enable Elevator_R input port

`off` (default) | `on`

Select this parameter to enable the **Elevator_R** input port.

Programmatic Use
Block Parameter: `Elevator_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left aileron rotation — Option to enable Left_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Aileron_R** input port.

Programmatic Use
Block Parameter: `Left_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right aileron rotation — Option to enable Right_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Aileron_R** input port.

Programmatic Use
Block Parameter: `Right_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Flaps rotation — Option to enable Flaps_R input port

`off` (default) | `on`

Select this parameter to enable the **Flaps_R** input port.

Programmatic Use
Block Parameter: `Flaps_R`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Gear

Nose wheel strut rotation — Option to enable Nose_Wheel_Strut_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_Strut_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_Strut_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel rotation — Option to enable Nose_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel rotation — Option to enable Left_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_R** input port.

Programmatic Use
Block Parameter: `Left_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel rotation — Option to enable Right_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_R** input port.

Programmatic Use
Block Parameter: `Right_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

NED (default) | ECEF

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use
Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History
Introduced in R2023b

R2024a: Simulation 3D Airliner Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame
Behavior changed in R2024a

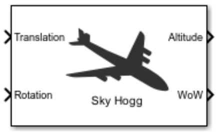
To enable ECEF coordinate systems for translation input, the Simulation 3D Airliner Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also
Simulation 3D Aircraft | Simulation 3D Scene Configuration

Simulation 3D Aircraft

Implement aircraft in 3D environment



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

Note Simulating models with the Simulation 3D Aircraft block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Aircraft to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Aircraft block implements an aircraft in a 3D visualization environment using translation and rotation to place the aircraft.

To use this block, ensure that the Simulation 3D Scene Configuration block is in your model. If you set the **Sample time** parameter of this block to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

The block input uses the aircraft north-east-down (NED) *right-handed* (RH) *Cartesian* coordinate system.

- *X*-axis — Along aircraft longitudinal axis, points forward
- *Y*-axis — Along aircraft lateral axis, points to the right
- *Z*-axis — Points downward

For more information, see “About Aerospace Coordinate Systems” on page 2-8.

Tip Verify that the Simulation 3D Aircraft block executes before the Simulation 3D Scene Configuration block. That way, Simulation 3D Aircraft prepares the signal data before the Unreal Engine 3D visualization environment receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Aircraft — `-1`

For more information about execution order, see “Control and Display Execution Order”.

Skeletons, Bones, and Meshes

Unreal uses a skeleton, bones, and mesh to define a 3D model. A skeleton is comprised of a set of bones. A mesh is the outer covering of the skeleton. Aircraft parts are sections of the mesh, such as ailerons or wheels, which are linked to the bones. For more information, see https://docs.unrealengine.com/4.27/AnimatingObjects/SkeletalMeshAnimation/Skeleton/.

For more information on how the Simulation 3D Aircraft block translation input arrays connect to aircraft types, see “Algorithms” on page 5-897.

Ports

Input

Translation — Aircraft translation
11-by-3 array | 12-by-3 array | 15-by-3 array | 30-by-3 array | 57-by-3 array

Aircraft translation, specified as:

- 11-by-3 array — Aircraft **Type** is `Sky Hogg`.
- 12-by-3 array — Aircraft **Type** is `Airliner`.
- 15-by-3 array — Aircraft **Type** is `General Aviation`.
- 30-by-3 array — Aircraft **Type** is `Air Transport`.
- 57-by-3 array — Aircraft **Type** is `Custom`.

The signal contains translation [X, Y, Z], in meters, with one row of the array for each bone of the aircraft.

The translation applies to these bones of the `Airliner` type:

Bone	Index
BODY	1
LEFT_ENGINE	2
RIGHT_ENGINE	3
RUDDER	4
ELEVATOR	5
LEFT_AILERON	6
RIGHT_AILERON	7
FLAPS	8
NOSE_WHEEL_STRUT	9
NOSE_WHEEL	10
LEFT_WHEEL	11
RIGHT_WHEEL	12

The translation applies to these bones of the `Sky Hogg` type:

Bone	Index
BODY	1
PROPELLER	2

Bone	Index
RUDDER	3
ELEVATOR	4
LEFT_AILERON	5
RIGHT_AILERON	6
FLAPS	7
NOSE_WHEEL_STRUT	8
NOSE_WHEEL	9
LEFT_WHEEL	10
RIGHT_WHEEL	11

The translation applies to these bones of the `General Aviation` type:

Bone	Index
BODY	1
ENGINE1	2
RUDDER	3
ELEVATOR	4
LEFT_AILERON	5
RIGHT_AILERON	6
FLAPS	7
LEFT_SPOILER	8
RIGHT_SPOILER	9
NOSE_GEAR	10
NOSE_WHEEL	11
LEFT_GEAR	12
LEFT_WHEEL	13
RIGHT_GEAR	14
RIGHT_WHEEL	15

The translation applies to these bones of the `Air Transport` type:

Bone	Index
BODY	1
ENGINE1	2
ENGINE2	3
ENGINE3	4
ENGINE4	5
RUDDER	6
ELEVATOR	7

Bone	Index
HORIZONTAL_STAB	8
LEFT_AILERON	9
RIGHT_AILERON	10
FLAPS1	11
FLAPS2	12
LEFT_SPOILER	13
RIGHT_SPOILER	14
NOSE_GEAR	15
NOSE_WHEEL	16
NOSE_GEAR_DOOR1	17
NOSE_GEAR_DOOR2	18
LEFT_GEAR	19
LEFT_WHEEL	20
LEFT_GEAR_LINK	21
LEFT_GEAR_DOOR1	22
LEFT_GEAR_DOOR2	23
LEFT_GEAR_DOOR2_2	24
RIGHT_GEAR	25
RIGHT_WHEEL	26
RIGHT_GEAR_LINK	27
RIGHT_GEAR_DOOR1	28
RIGHT_GEAR_DOOR2	29
RIGHT_GEAR_DOOR2_2	30

The translation applies to these bones of the `Custom` type:

Bone	Index
`BODY`	1
`ENGINE1`	2
`ENGINE1_PROP`	3
`ENGINE2`	4
`ENGINE2_PROP`	5
`ENGINE3`	6
`ENGINE3_PROP`	7
`ENGINE4`	8
`ENGINE4_PROP`	9
`ENGINE5`	10
`ENGINE5_PROP`	11

Bone	Index
ENGINE6	12
ENGINE6_PROP	13
ENGINE7	14
ENGINE7_PROP	15
ENGINE8	16
ENGINE8_PROP	17
ENGINE9	18
ENGINE9_PROP	19
ENGINE10	20
ENGINE10_PROP	21
ENGINE11	22
ENGINE11_PROP	23
ENGINE12	24
ENGINE12_PROP	25
ENGINE13	26
ENGINE13_PROP	27
ENGINE14	28
ENGINE14_PROP	29
ENGINE15	30
ENGINE15_PROP	31
ENGINE16	32
ENGINE16_PROP	33
WING1	34
WING1_LEFT_FLAP	35
WING1_RIGHT_FLAP	36
WING1_LEFT_AILERON	37
WING1_RIGHT_AILERON	38
WING1_LEFT_SPOILER	39
WING1_RIGHT_SPOILER	40
WING2	41
WING2_LEFT_FLAP	42
WING2_RIGHT_FLAP	43
HORIZONTAL_STABILIZER	44
LEFT_ELEVATOR	45
RIGHT_ELEVATOR	46
LEFT_RUDDER	47

Bone	Index
RIGHT_RUDDER	48
NOSE_GEAR	49
NOSE_WHEEL	50
NOSE_GEAR_DOOR	51
LEFT_GEAR	52
LEFT_WHEEL	53
LEFT_GEAR_DOOR	54
RIGHT_GEAR	55
RIGHT_WHEEL	56
RIGHT_GEAR_DOOR	57

Data Types: `single` | `double`

Rotation — Aircraft and wheel rotation
11-by-3 array | 12-by-3 array | 15-by-3 array | 30-by-3 array | 57-by-3 array

Aircraft rotation, specified as:

- 11-by-3 array — Aircraft **Type** is `Sky Hogg`.
- 12-by-3 array — Aircraft **Type** is `Airliner`.
- 15-by-3 array — Aircraft **Type** is `General Aviation`.
- 30-by-3 array — Aircraft **Type** is `Air Transport`.
- 57-by-3 array — Aircraft **Type** is `Custom`.

The rotation applies to the same bones as listed for the "Translation" on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the aircraft.

Data Types: `single` | `double`

LightStates — Aircraft light control
1-by-7 vector of double values

Aircraft light control, specified as a 1-by-7 vector of double values. Each element of the vector turns on (`1`) or off (`0`) a specific aircraft light group. The vector has this order:

- `LANDING_LIGHTS`
- `TAXI_LIGHTS`
- `ANTICOLLISION_BEACONS`
- `WINGTIP_STROBE_LIGHTS`
- `TAIL_STROBE_LIGHTS`
- `NAVIGATION_LIGHTS`
- `POSITION_LIGHTS`

Dependencies

To enable this port, set the **Light Configuration** parameter to `Configurable lights`.

Data Types: `single` | `double`

Output

Altitude — Aircraft attitude
1-by-4 vector

Aircraft altitude, returned as a 1-by-4 vector. The four altitudes are, in order:

- `aircraft_body`
- `aircraft_front_tire`
- `aircraft_left_tire`
- `aircraft_right_tire`

Dependencies

To enable this port, select the **Enable altitude sensor** check box.

Data Types: `double`

WoW — Weight on wheels
`1` | `0`

Aircraft weight on wheels logical switch, returned as `1`(true) if either of the contact locations or main gear tires (left or right) are on the ground. Otherwise, `0` (false) is returned.

Dependencies

To enable this port, select the **Enable altitude and Wow sensors** check box.

Data Types: `Boolean`

Parameters

Sample time — Sample time

`-1` (default) | real scalar

Sample time, T_s. The graphics frame rate is the inverse of the sample time.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: real scalar
Default: `'-1'`

Aircraft

Type — Aircraft type

`Sky Hogg` (default) | `Airliner` | `General Aviation` | `Air Transport` | `Custom`

Aircraft type, specified as `Sky Hogg`, `Airliner`, `General Aviation`, `Air Transport`, or `Custom`.

Dependencies

Setting this parameter requires that you set the **Initial translation** and **Initial rotation** parameters to the matching array size. Failure to appropriately set these array sizes causes an error.

Programmatic Use

Block Parameter: `Mesh`
Type: character vector
Values: `'Sky Hogg'` | `'Airliner'` | `'General Aviation'` | `'Air Transport'` | `'Custom'`
Default: `'Sky Hogg'`

Path to general aviation mesh — Path to general aviation mesh

`/MathWorksAerospaceContent/Vehicles/Aircraft/GeneralAviation/Mesh/SK_GeneralAviation.SK_GeneralAviation'` (default) | character vector

Path to general aviation mesh, specified as a character vector.

Dependencies

- To enable this parameter, set **Type** to `General Aviation`.

Programmatic Use

Block Parameter: `MeshPathGA`
Type: character vector
Values: `'/MathWorksAerospaceContent/Vehicles/Aircraft/GeneralAviation/Mesh/SK_GeneralAviation.SK_GeneralAviation'`
Default: `'/MathWorksAerospaceContent/Vehicles/Aircraft/GeneralAviation/Mesh/SK_GeneralAviation.SK_GeneralAviation'`

Path to air transport mesh — Path to air transport mesh

`/MathWorksAerospaceContent/Vehicles/Aircraft/AirTransport/Mesh/SK_AirTransport.SK_AirTransport` (default) | character vector

Path to air transport mesh, specified as a character vector.

Dependencies

- To enable this parameter, set **Type** to `Air Transport`.

Programmatic Use

Block Parameter: `MeshPathAT`
Type: character vector
Values: `'/MathWorksAerospaceContent/Vehicles/Aircraft/AirTransport/Mesh/SK_AirTransport.SK_AirTransport'`
Default: `'/MathWorksAerospaceContent/Vehicles/Aircraft/AirTransport/Mesh/SK_AirTransport.SK_AirTransport'`

Path to custom mesh — Path to custom mesh

`/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_HL20.SK_HL20` (default) | `/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_Aircraft.SK_Aircraft` | character vector

Path to custom mesh, specified as a character vector.

Dependencies

- To enable this parameter, set **Type** to `Custom`.

Programmatic Use

Block Parameter: `MeshPath`
Type: character vector
Values: `'/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_HL20.SK_HL20'` | `'/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_Aircraft.SK_Aircraft'`
Default: `'/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_HL20.SK_HL20'`

Color — Aircraft color

`Red` (default) | `Orange` | `Yellow` | `Green` | `Cyan` | `Blue` | `Black` | `White` | `Silver` | `Metal`

Aircraft color, specified as `Red`, `Orange`, `Yellow`, `Green`, `Cyan`, `Blue`, `Black`, `White`, `Silver`, or `Metal`.

Programmatic Use

Block Parameter: `AircraftColor`
Type: character vector
Values: `'Red'` | `'Orange'` | `'Yellow'` | `'Green'` | `'Cyan'` | `'Blue'` | `'Black'` | `'White'` | `'Silver'` | `'Metal'`
Default: `'Red'`

Name — Aircraft name

`SimulinkVehicle1` (default) | character vector

Aircraft name, specified as a character vector. By default, when you use the block in your model, the block sets the **Name** parameter to `SimulinkVehicleX`. The value of *X* depends on the number of other simulation 3D blocks that you have in your model.

Programmatic Use

Block Parameter: `ActorName`
Type: character vector
Values: scalar
Default: `'SimulinkVehicle1'`

Initial translation (m) — Initial translation of aircraft

`zeros(11,3)` (default) | 11-by-3 array | 12-by-3 array | 15-by-3 array | 30-by-3 array | 57-by-3 array

Initial translation of aircraft, specified as an 11-by-3, 12-by-3, 15-by-3, 30-by-3, or 57-by-3 array.

Dependencies

This parameter must match the aircraft type you set in **Type**. Failure to appropriately set these array sizes causes an error.

Programmatic Use

Block Parameter: `Translation`

Type: character vector
Values: 11-by-3 array | 12-by-3 array | 15-by-3 array | 30-by-3 array | 57-by-3 array
Default: `'zeros(11,3)'`

Data Types: `single` | `double`

Initial rotation (rad) — Aircraft rotation

`zeros(11,3)` (default) | 11-by-3 array | 12-by-3 array | 15-by-3 array | 30-by-3 array | 57-by-3 array

Initial rotation of aircraft, specified as a 11-by-3, 12-by-3, 15-by-3, 30-by-3, or 57-by-3 array.

Programmatic Use
Block Parameter: `Rotation`
Type: character vector
Values: 11-by-3 array | 12-by-3 array | 15-by-3 array | 30-by-3 array | 57-by-3 array
Default: `'zeros(11,3)'`

Data Types: `single` | `double`

Altitude Sensor

Enable altitude and WoW sensors — Altitude and WoW sensors

on (default) | off

To enable the altitude and WoW (weight on wheels) sensors, select this check box. Otherwise, clear this check box.

- Altitude sensors return the vertical distance to the ground below a vehicle origin, mimicking a radar altimeter. For more information, see “Altitude Sensors” on page 5-899.
- The WoW sensor returns a logical true (`1`) if either of the main landing gear wheels are on the ground, and false (`0`) otherwise. For more information, see “WoW Sensors” on page 5-900.

Programmatic Use
Block Parameter: `IsGHSensorEnabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Show sensor rays in viewer — Option to show sensor rays

off (default) | on

To show sensor rays in the viewer, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, select the **Enable altitude and Wow sensors** check box.

Programmatic Use
Block Parameter: `AreGHRaysVisible`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Interpret vector outputs as 1-D — Control reshaping on sensor output ports

off (default) | on

Control if reshaping occurs on the **Altitude WoW** sensor output ports:

- Select this check box to enable reshaping.
- Clear this check box to disable reshaping.

Programmatic Use
Block Parameter: `IsOutputReshapeEnabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Length of rays (m) — Length of rays

`1524` (default) | real scalar

Length of rays, specified as a real scalar in meters. The length of the rays limits the altitude detection. For example, if the vertical distance to the ground beneath the aircraft origin is greater than the length of the rays plus the aircraft body *Z* offset, the altitude sensor returns `-1` for the first value.

Dependencies

To enable this parameter, select the **Enable altitude sensor** check box.

Programmatic Use
Block Parameter: `GHRayLength`
Type: character vector
Values: real scalar
Default: `'1524'`

Body Z offset (m) — Aircraft body *Z* offset

`0.90` (default) | real scalar

Aircraft body *Z* offset, specified as a real scalar in meters.

Dependencies

To enable this parameter, select the **Enable altitude sensor** check box.

Programmatic Use
Block Parameter: `GHBodyOffset`
Type: character vector
Values: real scalar
Default: `'0.90'`

Front tire radius (m) — Front gear tire radius

`0.21` (default) | real scalar

Front gear tire radius, specified as a real scalar in meters. The front gear altitude ray originates at the front gear axle center plus the front gear tire radius *Z* offset.

Dependencies

To enable this parameter, select the **Enable altitude sensor** check box.

Programmatic Use

Block Parameter: `GHFrontTireRadius`
Type: character vector
Values: real scalar
Default: `'0.21'`

Left tire radius (m) — Left gear tire radius

0.21 (default) | real scalar

Left gear tire radius, specified as a real scalar in meters.

Dependencies

To enable this parameter, select the **Enable altitude sensor** check box.

Programmatic Use

Block Parameter: `GHLeftTireRadius`
Type: character vector
Values: real scalar
Default: `'0.21'`

Right tire radius (m) — Right gear tire radius

0.21 (default) | real scalar

Right gear tire radius, specified as a real scalar in meters.

Dependencies

To enable this parameter, select the **Enable altitude sensor** check box.

Programmatic Use

Block Parameter: `GHRightTireRadius`
Type: character vector
Values: real scalar
Default: `'0.21'`

WoW sensor tolerance (m) — Wow sensor tolerance

`0.05` (default) | real scalar

Wow sensor tolerance, specified as a real scalar.

Dependencies

To enable this parameter, select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `WoWSensorTolerance`
Type: character vector
Values: real scalar
Default: `'0.05'`

Light Configuration

Light configuration — Light configurations options

`Automatic lights` (default) | `Configurable lights` | `Lights off`

Light configuration options

- `Automatic lights` — Use default aircraft lighting configuration that provides realistic pattern cycling.
- `Configurable lights` — Configure aircraft lighting parameters.
- `Lights off` — Turn off all aircraft lights.

Dependencies

- Setting this parameter to `Automatic lights` disables the configurability of all other **Light Configuration** parameters. The block uses the default parameter values for aircraft lighting values.
- Setting this parameter to `Configurable lights` enables the configurability of the **Light Configuration** parameters according to the aircraft type.
- Setting **Type** to these aircraft types and **Light Configuration** to `Configurable lights` enables the configurability of the lighting parameters in use for each aircraft.

Type	Lights	Light Parameters
`Airliner`	**Landing lights**	• **Landing lights intensity (cd)** • **Landing lights cone half angle (deg)**
	Taxi lights	• **Taxi lights intensity (cd)** • **Taxi lights cone half angle (deg)**
	Red/green navigation lights	**Navigation lights intensity**
	White strobe lights	• **Strobe lights intensity** • **Wingtip strobe period (s)** • **Wingtip strobe pulse width (% of period)** • **Tail strobe period (s)** • **Tail strobe pulse width (% of period)**
	Red beacon lights	• **Beacon lights intensity** • **Beacon period (s)** • **Beacon pulse width (% of period)**
`Sky Hogg`	**Landing lights**	• **Landing lights intensity (cd)** • **Landing lights cone half angle (deg)**
	Red/green navigation lights	**Navigation lights intensity**

Type	Lights	Light Parameters
	White strobe lights	• **Strobe lights intensity** • **Tail strobe period (s)** • **Tail strobe pulse width (% of period)**
	Red beacon lights	• **Beacon lights intensity** • **Beacon period (s)** • **Beacon pulse width (% of period)**
`General Aviation`, `Air Transport`, or `Custom`	**Landing lights**	• **Landing lights intensity (cd)** • **Landing lights cone half angle (deg)** • **Left landing light location** • **Left landing light orientation** • **Right landing light location** • **Right landing light orientation**
	Taxi lights	• **Taxi lights intensity (cd)** • **Taxi lights cone half angle (deg)** • **Taxi lights location** • **Taxi lights orientation (deg)**
	Red/green navigation lights	**Navigation lights intensity**
	White navigation lights	**Position light intensity**
	White strobe lights	• **Strobe lights intensity** • **Wingtip strobe period (s)** • **Wingtip strobe pulse width (% of period)** • **Tail strobe period (s)** • **Tail strobe pulse width (% of period)**
	Red beacon lights	• **Beacon lights intensity** • **Beacon period (s)** • **Beacon pulse width (% of period)**

- Setting this parameter to `Lights off` disables the configurability of all other **Light Configuration** parameters. The block turns off all aircraft lighting.

Programmatic Use
Block Parameter: `LightsConfig`
Type: character vector
Values: `'Automatic lights'` | `'Configurable lights'` | `'Lights off'`
Default: `'Automatic lights'`

Landing lights intensity (cd) — Landing lights intensity

30000 (default) | positive scalar

Landing lights intensity, specified as a positive scalar, in candela.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `LandingLightIntensity`
Type: character vector
Values: positive scalar
Default: `'30000'`

Landing lights cone half angle (deg) — Landing lights cone half angle

15 (default) | positive scalar

Landing lights cone half angle, specified as a positive scalar, in degrees.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `LandingLightConeAngle`
Type: character vector
Values: positive scalar
Default: `'15'`

Left landing light location — Left landing light location

`[0 0 0]` (default) | 3-element vector

Left landing light location with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Type** to `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `CustomLeftLandingLightLocation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Left landing light orientation (deg) — Left landing light orientation

`[0 0 0]` (default) | 3-element vector

Left landing light orientation with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Type** to `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `CustomLeftLandingLightOrientation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Right landing light location — Right landing light location

`[0 0 0]` (default) | 3-element vector

Right landing light location with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Type** to `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `CustomRightLandingLightLocation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Right landing light orientation (deg) — Right landing light orientation

`[0 0 0]` (default) | 3-element vector

Right landing light orientation with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Type** to `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `CustomRightLandingLightOrientation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Taxi lights intensity (cd) — Taxi lights intensity

`150000` (default) | positive scalar

Taxi lights intensity, specified as a positive scalar, in candela.

Dependencies

To enable this parameter, set **Type** to `Airliner`, `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `TaxiLightIntensity`
Type: character vector

Values: positive scalar
Default: `'150000'`

Taxi lights cone half angle (deg) — Taxi lights cone half angle

36 (default) | positive scalar

Taxi lights cone half angle, specified as a positive scalar, in degrees.

Dependencies

To enable this parameter, set **Type** to `Airliner`, `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `TaxiLightConeAngle`
Type: character vector
Values: positive scalar
Default: `'36'`

Taxi lights location — Taxi lights location

`[0 0 0]` (default) | 3-element vector

Taxi lights location with respect to the associated bone of the skeletal mesh, specified as a 3-element vector

Dependencies

To enable this parameter, set **Type** to `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `CustomTaxiLightLocation`
Type: character vector
Values: positive scalar
Default: `'[0 0 0]'`

Taxi lights orientation (deg) — Taxi lights orientation

`[0 0 0]` (default) | 3-element vector

Taxi lights orientation with respect to the associated bone of the skeletal mesh, specified as a 3-element vector

Dependencies

To enable this parameter, set **Type** to `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `CustomTaxiLightOrientation`
Type: character vector
Values: positive scalar
Default: `'[0 0 0]'`

Navigation lights intensity — Navigation lights intensity

`500` (default) | positive scalar

Navigation lights intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `NavLightIntensity`
Type: character vector
Values: positive scalar
Default: `'500'`

Position light intensity — Position light intensity

`500` (default) | positive scalar

Position light intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `PositionLightIntensity`
Type: character vector
Values: positive scalar
Default: `'500'`

Strobe lights intensity — Strobe lights intensity

`5000` (default) | positive scalar

Strobe lights intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `StrobeLightIntensity`
Type: character vector
Values: positive scalar
Default: `'5000'`

Wingtip strobe period (s) — Wingtip strobe period

`1.5` (default) | positive scalar

Wingtip strobe period, specified as a positive scalar.

Dependencies

To enable this parameter, set **Type** to `Airliner`, `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `WingtipStrobePeriod`
Type: character vector
Values: positive scalar
Default: `'1.5'`

Wingtip strobe pulse width (% of period) — Wingtip strobe pulse width

6 (default) | positive scalar

Wingtip strobe pulse width, specified as a positive scalar.

Dependencies

To enable this parameter, set **Type** to `Airliner`, `General Aviation`, `Air Transport`, or `Custom` and **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `WingtipStrobePulseWidth`
Type: character vector
Values: positive scalar
Default: `'6'`

Tail strobe period (s) — Tail strobe period

`1.5` (default) | positive scalar

Tail strobe period, specified as a positive scalar, in seconds.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `TailStrobePeriod`
Type: character vector
Values: positive scalar
Default: `'1.5'`

Tail strobe pulse width (% of period) — Tail strobe pulse width

6 (default) | positive scalar

Tail strobe pulse width, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `TailStrobePulseWidth`
Type: character vector
Values: positive scalar
Default: `'6'`

Beacon lights intensity — Beacon lights intensity

`4000` (default) | positive scalar

Beacon lights intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconLightIntensity`
Type: character vector
Values: positive scalar
Default: `'4000'`

Beacon period (s) — Beacon period

`1.5` (default) | positive scalar

Beacon period, specified as a positive scalar, in seconds.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconPeriod`
Type: character vector
Values: positive scalar
Default: `'1.5'`

Beacon pulse width (% of period) — Beacon pulse width

`10` (default) | positive scalar

Beacon pulse width, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconPulseWidth`
Type: character vector
Values: positive scalar
Default: `'10'`

Algorithms

Airliner and Sky Hogg Aircraft Types

This topic lists how the block input arrays, 11-by-3, 12-by-3, 15-by-3, 30-by-3, and 57-by-3, connect to their associated aircraft types.

In Unreal skeletons, all bones have six degrees of freedom. However, for the Airliner and Sky Hogg aircraft types, the Simulation 3D Aircraft block only enables all six degrees of freedom (6DOF) for the BODY aircraft bone. For the other aircraft bones, the block enables only one degree of freedom. For more information, see Airliner Active Degrees of Freedom and Sky Hogg Active Degrees of Freedom.

In these tables, the markings indicate enabled or disabled degrees of freedom for corresponding aircraft bones. Columns **X**, **Y**, **Z**, **Pitch**, **Roll**, **Pitch**, and **Yaw** each correspond to one degree for the associated bone.

- ✓ — Degree of freedom enabled for the aircraft bone.
- X — Degree of freedom disabled for the aircraft bone.

Airliner Active Degrees of Freedom

Bone	Index	X	Y	Z	Roll	Pitch	Yaw
BODY	1	✓	✓	✓	✓	✓	✓
LEFT_ENGINE	2	X	X	X	✓	X	X
RIGHT_ENGINE	3	X	X	X	✓	X	X
RUDDER	4	X	X	X	X	X	✓
ELEVATOR	5	X	X	X	X	✓	X
LEFT_AILERON	6	X	X	X	X	✓	X
RIGHT_AILERON	7	X	X	X	X	✓	X
FLAPS	8	X	X	X	X	✓	X
NOSE_WHEEL_STRUT	9	X	X	X	X	X	✓
NOSE_WHEEL	10	X	X	X	X	✓	X
LEFT_WHEEL	11	X	X	X	X	✓	X
RIGHT_WHEEL	12	X	X	X	X	✓	X

Sky Hogg Active Degrees of Freedom

Bone	Index	X	Y	Z	Roll	Pitch	Yaw
BODY	1	✓	✓	✓	✓	✓	✓
PROPELLER	2	X	X	X	✓	X	X
RUDDER	3	X	X	X	X	X	✓
ELEVATOR	4	X	X	X	X	✓	X
LEFT_AILERON	5	X	X	X	X	✓	X
RIGHT_AILERON	6	X	X	X	X	✓	X
FLAPS	7	X	X	X	X	✓	X
NOSE_WHEEL_STRUT	8	X	X	X	X	X	✓
NOSE_WHEEL	9	X	X	X	X	✓	X
LEFT_WHEEL	10	X	X	X	X	✓	X
RIGHT_WHEEL	11	X	X	X	X	✓	X

General Aviation, Air Transport, and Custom Aircraft Types

For general aviation, air transport and custom aircraft, the Simulation 3D Aircraft block enables six degrees of freedom (6DOF) for all aircraft bones. Note, the default poses for each bone in the skeletal mesh affects how the mesh deforms when you manipulate each degree of freedom.

Altitude Sensors

Altitude sensors return the vertical distance to the ground below a vehicle origin, mimicking a radar altimeter. The Simulation 3D Aircraft block determines the vertical distance by calculating the starting and ending locations of the ray.

- Starting location

  The block calculates the starting location of the ray using the equation:

  starting_location = *representative_bone* - *Z* offset specified in block

Representative Bone	Z Offset Parameter
`BODY`	**Body Z offset**
`NOSE_WHEEL`	**Front tire radius**
`LEFT_WHEEL`	**Left tire radius**
`RIGHT_WHEEL`	**Right tire radius**

- Ending location

  The block calculates the ending location using the equation:

  ending_location = starting_location - **Length of rays** or the point of first contact with a surface in the Unreal scene.

How Altitude Sensors Work

The block determines the altitude with this workflow:

1. The block gives the ray starting point, direction, and maximum length to the altitude sensor, which exists in an Unreal Engine scene.
2. The Unreal Engine projects the ray from the starting point in the specified direction and determines if an object is within **Length of rays**.
3. Unreal Engine returns to the block information such as:
   - If an object is detected or not
   - If an object is detected, the location of the object
4. Using this information, the block calculate the altitude sensor values.

WoW Sensors

Aircraft use WoW switches to enable and disable systems. Weight-on-Wheels (WoW) sensors return a logical value of `1` (true) if one or more of the landing gear wheels touches a surface, `0` (false) otherwise. For example, use thrust reversers only when the WoW sensor returns true or allow landing gear retraction only when the WoW sensor returns `0` (false).

To determine if any main landing gear tires touches the ground or any object, the WoW sensor uses the third and fourth rays of the altitude sensor. The left and right tire radius offsets must equal the actual radii of the tire meshes.

- If the sensed ground location is within ±**WoW sensor tolerance** of a ground contact point, the block returns a WoW sensor output of `1` (logical true).
- Otherwise, the block returns `0` (logical false).

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

Simulating models with the Simulation 3D Aircraft block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Aircraft to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

R2024a: Simulation 3D Aircraft Block Supports Single-Precision Inputs

Behavior changed in R2024a

The Simulation 3D Aircraft block now supports single- and double-precision inputs for the **Translation** and **Rotation** input ports. Prior to R2024a, this block supported only double-precision inputs for these ports.

R2023a: Simulation 3D Aircraft Block Updates

Behavior changed in R2023a

The Simulation 3D Aircraft block has these changes:

- The contents of the block **Initial Values** tab have moved to the **Aircraft** tab under the **Initial Values** section.
- The **Altitude Sensor** tab has these changes:

  - **Enable altitude sensor** has been renamed to **Enable Altitude and WoW sensors**.

    Starting with R2023a, the WoW (weight on wheels) sensor uses the altitude sensor rays and offsets that you specify in the block mask to determine if one or more wheel or ground contact location is touching the ground.
  - **Enable visible sensor rays** has been renamed to **Show sensor rays in viewer**.
  - **Aircraft body Z offset (in meters)** has been renamed to **Body Z offset (m)**.
  - **Front gear tire radius (in meters)** has been renamed to **Front tire radius (m)**.
  - **Left gear tire radius (in meters)** has been renamed to **Left tire radius (m)**.
  - **Right gear tire radius (in meters)** has been renamed to **Right tire radius (m)**.
  - **WoW sensor tolerance**, to set a WoW sensor tolerance.
  - New parameter, **Interpret vector outputs as 1-D**, to control if reshaping occurs on the **Altitude** and **WoW** sensor output ports.

R2022a: Support for Custom Meshes and Aircraft Lighting

Behavior changed in R2022a

The block now supports:

- The specification of a custom aircraft mesh with the **Path to custom mesh** parameter. To enable this parameter, set **Type** to `Custom`.
- The configuration of aircraft lighting with a new **Light Configuration** tab.
- More colors for aircraft bodies in the **Color** parameter.

See Also

Blocks

Simulation 3D Actor Transform Get | Simulation 3D Actor Transform Set | Simulation 3D Camera Get | Simulation 3D Scene Configuration | Simulation 3D Message Get | Simulation 3D Message Set | Simulation 3D Rotorcraft

Tools

Airliner | Air Transport | Custom | General Aviation | Sky Hogg

Simulation 3D Air Transport Pack

Generate translation and rotation information for air transport aircraft

Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Air Transport Pack block creates translation and rotation information for the Simulation 3D Aircraft block with **Type** set to `Air transport`. Use the Simulation 3D Air Transport Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Aircraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_T — Engine 1 translation
1-by-3 matrix

Engine 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 translation** parameter.

Data Types: `single` | `double`

Engine2_T — Engine 2 translation
1-by-3 matrix

Engine 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 translation** parameter.

Data Types: `single` | `double`

Engine3_T — Engine 3 translation
1-by-3 matrix

Engine 3 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 3 translation** parameter.

Data Types: `single` | `double`

Engine4_T — Engine 4 translation
1-by-3 matrix

Engine 4 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 4 translation** parameter.

Data Types: `single` | `double`

Rudder_T — Rudder translation
1-by-3 matrix

Rudder translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rudder translation** parameter.

Data Types: `single` | `double`

Elevator_T — Elevator translation
1-by-3 matrix

Elevator translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Elevator translation** parameter.

Data Types: `single` | `double`

Horizontal_Stab_T — Horizontal stab translation
1-by-3 matrix

Horizontal stab translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Horizontal stab translation** parameter.

Data Types: `single` | `double`

Left_Aileron_T — Left aileron translation
1-by-3 matrix

Left aileron translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left aileron translation** parameter.

Data Types: `single` | `double`

Right_Aileron_T — Right aileron translation
1-by-3 matrix

Right aileron translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right aileron translation** parameter.

Data Types: `single` | `double`

Flaps1_T — Flaps set 1 translation
1-by-3 matrix

Flaps set 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps set 1 translation** parameter.

Data Types: `single` | `double`

Flaps2_T — Flaps set 2 translation
1-by-3 matrix

Flaps set 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps set 2 translation** parameter.

Data Types: `single` | `double`

Left_Spoiler_T — Left spoiler translation
1-by-3 matrix

Left spoiler translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left spoiler translation** parameter.

Data Types: `single` | `double`

Right_Spoiler_T — Right spoiler translation
1-by-3 matrix

Right spoiler translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right spoiler translation** parameter.

Data Types: `single` | `double`

Nose_Gear_T — Nose gear translation
1-by-3 matrix

Nose gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear translation** parameter.

Data Types: `single` | `double`

Nose_Wheel_T — Nose wheel translation
1-by-3 matrix

Nose wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel translation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door1_T — Nose gear door 1 translation
1-by-3 matrix

Nose gear door 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 1 translation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door2_T — Nose gear door 2 translation
1-by-3 matrix

Nose gear door 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 2 translation** parameter.

Data Types: `single` | `double`

Left_Gear_T — Left gear translation
1-by-3 matrix

Left gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear translation** parameter.

Data Types: `single` | `double`

Left_Wheel_T — Left wheel translation
1-by-3 matrix

Left wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel translation** parameter.

Data Types: `single` | `double`

Left_Gear_Link_T — Left gear link translation
1-by-3 matrix

Left gear link translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear link translation** parameter.

Data Types: `single` | `double`

Left_Gear_Door1_T — Left gear door 1 translation
1-by-3 matrix

Left gear door 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 1 translation** parameter.

Data Types: `single` | `double`

Left_Gear_Door2_T — Left gear door 2 translation
1-by-3 matrix

Left gear door 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 2 translation** parameter.

Data Types: `single` | `double`

Left_Gear_Door2_2_T — Left gear door 2.2 translation
1-by-3 matrix

Left gear door 2.2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 2.2 translation** parameter.

Data Types: `single` | `double`

Right_Gear_T — Right gear translation
1-by-3 matrix

Right gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear translation** parameter.

Data Types: `single` | `double`

Right_Wheel_T — Right wheel translation
1-by-3 matrix

Right wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel translation** parameter.

Data Types: `single` | `double`

Right_Gear_Link_T — Right gear link translation
1-by-3 matrix

Right gear link translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear link translation** parameter.

Data Types: `single` | `double`

Right_Gear_Door1_T — Right gear door 1 translation
1-by-3 matrix

Right gear door 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 1 translation** parameter.

Data Types: `single` | `double`

Right_Gear_Door2_T — Right gear door 2 translation
1-by-3 matrix

Right gear door 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 2 translation** parameter.

Data Types: `single` | `double`

Right_Gear_Door2_2_T — Right gear door 2.2 translation
1-by-3 matrix

Right gear door 2.2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 2.2 translation** parameter.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_R — Engine 1 rotation
1-by-3 matrix

Engine 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 rotation** parameter.

Data Types: `single` | `double`

Engine2_R — Engine 2 rotation
1-by-3 matrix

Engine 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 rotation** parameter.

Data Types: `single` | `double`

Engine3_R — Engine 3 rotation
1-by-3 matrix

Engine 3 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 3 rotation** parameter.

Data Types: `single` | `double`

Engine4_R — Engine 4 rotation
1-by-3 matrix

Engine 4 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 4 rotation** parameter.

Data Types: `single` | `double`

Rudder_R — Rudder rotation
1-by-3 matrix

Rudder rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rudder rotation** parameter.

Data Types: `single` | `double`

Elevator_R — Elevator rotation
1-by-3 matrix

Elevator rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Elevator rotation** parameter.

Data Types: `single` | `double`

Horizontal_Stab_R — Horizontal stab rotation
1-by-3 matrix

Horizontal stab rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Horizontal stab rotation** parameter.

Data Types: `single` | `double`

Left_Aileron_R — Left aileron rotation
1-by-3 matrix

Left aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left aileron rotation** parameter.

Data Types: `single` | `double`

Right_Aileron_R — Right aileron rotation
1-by-3 matrix

Right aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right aileron rotation** parameter.

Data Types: `single` | `double`

Flaps1_R — Flaps set 1 rotation
1-by-3 matrix

Flaps set 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps set 1 rotation** parameter.

Data Types: `single` | `double`

Flaps2_R — Flaps set 2 rotation
1-by-3 matrix

Flaps set 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps set 2 rotation** parameter.

Data Types: `single` | `double`

Left_Spoiler_R — Left spoiler rotation
1-by-3 matrix

Left spoiler rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left spoiler rotation** parameter.

Data Types: `single` | `double`

Right_Spoiler_R — Right spoiler rotation
1-by-3 matrix

Right spoiler rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right spoiler rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_R — Nose gear rotation
1-by-3 matrix

Nose gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_R — Nose wheel rotation
1-by-3 matrix

Nose wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door1_R — Nose gear door 1 rotation
1-by-3 matrix

Nose gear door 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 1 rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door2_R — Nose gear door 2 rotation
1-by-3 matrix

Nose gear door 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 2 rotation** parameter.

Data Types: `single` | `double`

Left_Gear_R — Left gear rotation
1-by-3 matrix

Left gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear rotation** parameter.

Data Types: `single` | `double`

Left_Wheel_R — Left wheel rotation
1-by-3 matrix

Left wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel rotation** parameter.

Data Types: `single` | `double`

Left_Gear_Link_R — Left gear link rotation
1-by-3 matrix

Left gear link rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear link rotation** parameter.

Data Types: `single` | `double`

Left_Gear_Door1_R — Left gear door 1 rotation
1-by-3 matrix

Left gear door 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 1 rotation** parameter.

Data Types: `single` | `double`

Left_Gear_Door2_R — Left gear door 2 rotation
1-by-3 matrix

Left gear door 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 2 rotation** parameter.

Data Types: `single` | `double`

Left_Gear_Door2_2_R — Left gear door 2.2 rotation
1-by-3 matrix

Left gear door 2.2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 2.2 rotation** parameter.

Data Types: `single` | `double`

Right_Gear_R — Right gear rotation
1-by-3 matrix

Right gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear rotation** parameter.

Data Types: `single` | `double`

Right_Wheel_R — Right wheel rotation
1-by-3 matrix

Right wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel rotation** parameter.

Data Types: `single` | `double`

Right_Gear_Link_R — Right gear link rotation
1-by-3 matrix

Right gear link rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear link rotation** parameter.

Data Types: `single` | `double`

Right_Gear_Door1_R — Right gear door 1 rotation
1-by-3 matrix

Right gear door 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 1 rotation** parameter.

Data Types: `single` | `double`

Right_Gear_Door2_R — Right gear door 2 rotation
1-by-3 matrix

Right gear door 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 2 rotation** parameter.

Data Types: `single` | `double`

Right_Gear_Door2_2_R — Right gear door 2.2 rotation
1-by-3 matrix

Right gear door 2.2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 2.2 rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Aircraft translation
`30`-by-`3` array

Aircraft translation for air transport aircraft, returned as a `30`-by-`3` array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the aircraft.

The translation applies to these bones of the `Air transport` type.

Bone	Index
BODY	1
ENGINE1	2
ENGINE2	3
ENGINE3	4
ENGINE4	5
RUDDER	6
ELEVATOR	7
HORIZONTAL_STAB	8
LEFT_AILERON	9
RIGHT_AILERON	10
FLAPS1	11
FLAPS2	12
LEFT_SPOILER	13
RIGHT_SPOILER	14
NOSE_GEAR	15
NOSE_WHEEL	16
NOSE_GEAR_DOOR1	17
NOSE_GEAR_DOOR2	18
LEFT_GEAR	19
LEFT_WHEEL	20
LEFT_GEAR_LINK	21
LEFT_GEAR_DOOR1	22
LEFT_GEAR_DOOR2	23

Bone	Index
LEFT_GEAR_DOOR2_2	24
RIGHT_GEAR	25
RIGHT_WHEEL	26
RIGHT_GEAR_LINK	27
RIGHT_GEAR_DOOR1	28
RIGHT_GEAR_DOOR2	29
RIGHT_GEAR_DOOR2_2	30

Rotation — Aircraft rotation
30-by-3 array

Aircraft rotation for air transport aircraft, returned as a 30-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the aircraft.

Parameters

Propulsion

Engine 1 translation — Option to enable Engine1_T input port

off (default) | on

Select this parameter to enable the **Engine1_T** input port.

Programmatic Use
Block Parameter: Engine1_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Engine 2 translation — Option to enable Engine2_T input port

off (default) | on

Select this parameter to enable the **Engine2_T** input port.

Programmatic Use
Block Parameter: Engine2_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Engine 3 translation — Option to enable Engine3_T input port

off (default) | on

Select this parameter to enable the **Engine3_T** input port.

Programmatic Use
Block Parameter: `Engine3_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 4 translation — Option to enable Engine4_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine4_T** input port.

Programmatic Use
Block Parameter: `Engine4_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 1 rotation — Option to enable Engine1_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_R** input port.

Programmatic Use
Block Parameter: `Engine1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 rotation — Option to enable Engine2_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_R** input port.

Programmatic Use
Block Parameter: `Engine2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 3 rotation — Option to enable Engine3_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine3_R** input port.

Programmatic Use
Block Parameter: `Engine3_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 4 rotation — Option to enable Engine4_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine4_R** input port.

Programmatic Use
Block Parameter: `Engine4_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Controls

Rudder translation — Option to enable Rudder_T input port

`off` (default) | `on`

Select this parameter to enable the **Rudder_T** input port.

Programmatic Use
Block Parameter: `Rudder_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Elevator translation — Option to enable Elevator_T input port

`off` (default) | `on`

Select this parameter to enable the **Elevator_T** input port.

Programmatic Use
Block Parameter: `Elevator_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Horizontal stab translation — Option to enable Horizontal_Stab_T input port

`off` (default) | `on`

Select this parameter to enable the **Horizontal_Stab_T** input port.

Programmatic Use
Block Parameter: `Horizontal_Stab_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left aileron translation — Option to enable Left_Aileron_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Aileron_T** input port.

Programmatic Use
Block Parameter: `Left_Aileron_T`

Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right aileron translation — Option to enable Right_Aileron_T input port

off (default) | on

Select this parameter to enable the **Right_Aileron_T** input port.

Programmatic Use
Block Parameter: Right_Aileron_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Flaps set 1 translation — Option to enable Flaps1_T input port

off (default) | on

Select this parameter to enable the **Flaps1_T** input port.

Programmatic Use
Block Parameter: Flaps1_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Flaps set 2 translation — Option to enable Flaps2_T input port

off (default) | on

Select this parameter to enable the **Flaps2_T** input port.

Programmatic Use
Block Parameter: Flaps2_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Left spoiler translation — Option to enable Left_Spoiler_T input port

off (default) | on

Select this parameter to enable the **Left_Spoiler_T** input port.

Programmatic Use
Block Parameter: Left_Spoiler_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right spoiler translation — Option to enable Right_Spoiler_T input port

off (default) | on

Select this parameter to enable the **Right_Spoiler_T** input port.

Programmatic Use
Block Parameter: `Right_Spoiler_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rudder rotation — Option to enable Rudder_R input port

`off` (default) | `on`

Select this parameter to enable the **Rudder_R** input port.

Programmatic Use
Block Parameter: `Rudder_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Elevator rotation — Option to enable Elevator_R input port

`off` (default) | `on`

Select this parameter to enable the **Elevator_R** input port.

Programmatic Use
Block Parameter: `Elevator_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Horizontal stab rotation — Option to enable Horizontal_Stab_R input port

`off` (default) | `on`

Select this parameter to enable the **Horizontal_Stab_R** input port.

Programmatic Use
Block Parameter: `Horizontal_Stab_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left aileron rotation — Option to enable Left_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Aileron_R** input port.

Programmatic Use
Block Parameter: `Left_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right aileron rotation — Option to enable Right_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Aileron_R** input port.

Programmatic Use
Block Parameter: `Right_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Flaps set 1 rotation — Option to enable Flaps1_R input port

`off` (default) | `on`

Select this parameter to enable the **Flaps1_R** input port.

Programmatic Use
Block Parameter: `Flaps1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Flaps set 2 rotation — Option to enable Flaps2_R input port

`off` (default) | `on`

Select this parameter to enable the **Flaps2_R** input port.

Programmatic Use
Block Parameter: `Flaps2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left spoiler rotation — Option to enable Left_Spoiler_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Spoiler_R** input port.

Programmatic Use
Block Parameter: `Left_Spoiler_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right spoiler rotation — Option to enable Right_Spoiler_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Spoiler_R** input port.

Programmatic Use
Block Parameter: `Right_Spoiler_R`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Gear

Nose gear translation — Option to enable Nose_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel translation — Option to enable Nose_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_T** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 1 translation — Option to enable Nose_Gear_Door1_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_Door1_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 2 translation — Option to enable Nose_Gear_Door2_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear translation — Option to enable Left_Gear_T input port

off (default) | on

Select this parameter to enable the **Left_Gear_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel translation — Option to enable Left_Wheel_T input port

off (default) | on

Select this parameter to enable the **Left_Wheel_T** input port.

Programmatic Use
Block Parameter: `Left_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear link translation — Option to enable Left_Gear_Link_T input port

off (default) | on

Select this parameter to enable the **Left_Gear_Link_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_Link_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 1 translation — Option to enable Left_Gear_Door1_T input port

off (default) | on

Select this parameter to enable the **Left_Gear_Door1_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 2 translation — Option to enable Left_Gear_Door2_T input port

off (default) | on

Select this parameter to enable the **Left_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`

Default: 'off'

Left gear door 2.2 translation — Option to enable Left_Gear_Door2_2_T input port

off (default) | on

Select this parameter to enable the **Left_Gear_Door2_2_T** input port.

Programmatic Use
Block Parameter: Left_Gear_Door2_2_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right gear translation — Option to enable Right_Gear_T input port

off (default) | on

Select this parameter to enable the **Right_Gear_T** input port.

Programmatic Use
Block Parameter: Right_Gear_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right wheel translation — Option to enable Right_Wheel_T input port

off (default) | on

Select this parameter to enable the **Right_Wheel_T** input port.

Programmatic Use
Block Parameter: Right_Wheel_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right gear link translation — Option to enable Right_Gear_Link_T input port

off (default) | on

Select this parameter to enable the **Right_Gear_Link_T** input port.

Programmatic Use
Block Parameter: Right_Gear_Link_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right gear door 1 translation — Option to enable Right_Gear_Door1_T input port

off (default) | on

Select this parameter to enable the **Right_Gear_Door1_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 2 translation — Option to enable Right_Gear_Door2_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 2.2 translation — Option to enable Right_Gear_Door2_2_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door2_2_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door2_2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear rotation — Option to enable Nose_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel rotation — Option to enable Nose_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 1 rotation — Option to enable Nose_Gear_Door1_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_Door1_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 2 rotation — Option to enable Nose_Gear_Door2_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear rotation — Option to enable Left_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel rotation — Option to enable Left_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_R** input port.

Programmatic Use
Block Parameter: `Left_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear link rotation — Option to enable Left_Gear_Link_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_Link_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Link_R`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Left gear door 1 rotation — Option to enable Left_Gear_Door1_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_Door1_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 2 rotation — Option to enable Left_Gear_Door2_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_Door2_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 2.2 rotation — Option to enable Left_Gear_Door2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_Door2_2_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door2_2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear rotation — Option to enable Right_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel rotation — Option to enable Right_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_R** input port.

Programmatic Use
Block Parameter: `Right_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear link rotation — Option to enable Right_Gear_Link_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Link_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Link_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 1 rotation — Option to enable Right_Gear_Door1_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door1_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 2 rotation — Option to enable Right_Gear_Door2_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door2_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 2.2 rotation — Option to enable Right_Gear_Door2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door2_2_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door2_2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use
Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History

Introduced in R2023b

R2024a: Simulation 3D Air Transport Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame

Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D Air Transport Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

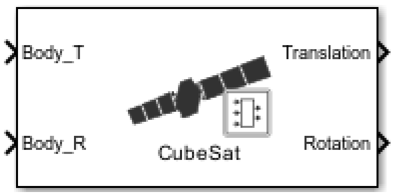
- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also

Simulation 3D Aircraft | Simulation 3D Scene Configuration

Simulation 3D CubeSat Pack

Create translation and rotation input matrices for CubeSat



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D CubeSat Pack block creates translation and rotation information for the Simulation 3D Spacecraft block with **Type** set to `CubeSat`. Use the Simulation 3D CubeSat Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Spacecraft block. BB

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Output

Translation — Spacecraft translation
10-by-3 array

Spacecraft translation for CubeSat, returned as an 10-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the spacecraft.

The translation applies to these bones of the `CubeSat` type.

Bone	Index
BODY	1
ANTENNA1	2

Bone	Index
ANTENNA2	3
ANTENNA3	4
SENSOR	5
SOLAR_ARRAY1	6
SOLAR_ARRAY1_1	7
SOLAR_ARRAY2	8
SOLAR_ARRAY2_2	9
THRUSTER	10

Rotation — Spacecraft rotation
10-by-3 array

Spacecraft rotation for CubeSat, returned as an 10-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the spacecraft.

Parameters

To edit block parameters interactively, use the Property Inspector. From the Simulink Toolstrip, on the **Simulation** tab, in the **Prepare** gallery, select **Property Inspector**.

Sensors

Antenna 1 translation — Option to enable Antenna1_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna1_T`
Values:	`'off'` (default) | `'on'`

Antenna 1 rotation — Option to enable Antenna1_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna1_R`
Values:	`'off'` (default) | `'on'`

Antenna 2 translation — Option to enable Antenna2_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna2_T`
Values:	`'off'` (default) | `'on'`

Antenna 2 rotation — Option to enable Antenna2_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna2_R`
Values:	`'off'` (default) | `'on'`

Antenna 3 translation — Option to enable Antenna3_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna3_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna3_T`
Values:	`'off'` (default) | `'on'`

Antenna 3 rotation — Option to enable Antenna3_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna3_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna3_R`
Values:	`'off'` (default) | `'on'`

Sensor translation — Option to enable Sensor_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor_T`
Values:	`'off'` (default) | `'on'`

Sensor rotation — Option to enable Sensor_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor_R`
Values:	`'off'` (default) | `'on'`

Solar Arrays

Solar array 1 translation — Option to enable SolarArray1_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1 rotation — Option to enable SolarArray1_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_R`
Values:	`'off'` (default) | `'on'`

Solar array 1.1 translation — Option to enable SolarArray1_1_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1.1 rotation — Option to enable SolarArray1_1_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_R`
Values:	`'off'` (default) | `'on'`

Solar array 2 translation — Option to enable SolarArray2_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_T`
Values:	`'off'` (default) | `'on'`

Solar array 2 rotation — Option to enable SolarArray2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_R`
Values:	`'off'` (default) | `'on'`

Solar array 2.2 translation — Option to enable SolarArray2_2_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_T`
Values:	`'off'` (default) | `'on'`

Solar array 2.2 rotation — Option to enable SolarArray2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_R`
Values:	`'off'` (default) | `'on'`

Thrusters

Thruster translation — Option to enable Thruster_T input port

`off` (default) | `on`

Select this parameter to enable the **Thruster_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Thruster_T`
Values:	`'off'` (default) | `'on'`

Thruster rotation — Option to enable Thruster_R input port

`off` (default) | `on`

Select this parameter to enable the **Thruster_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Thruster_R`
Values:	`'off'` (default) | `'on'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`AdjustForCesium`

Values:	`'off'` (default) | `'on'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Heading0`
Values:	`'90'` (default) | real scalar

Version History

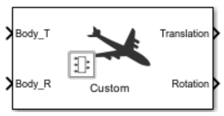
Introduced in R2024a

See Also

Simulation 3D Spacecraft | CubeSat | CubeSat Vehicle

Simulation 3D Custom Pack

Generate translation and rotation information for custom aircraft



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Custom Pack block creates translation and rotation information for the Simulation 3D Aircraft block with **Type** set to `Custom`. Use the Simulation 3D Custom Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Aircraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_T — Engine 1 translation
1-by-3 matrix

Engine 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 translation** parameter.

Data Types: `single` | `double`

Engine1_Prop_T — Engine 1 propeller translation
1-by-3 matrix

Engine 1 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 propeller translation** parameter.

Data Types: `single` | `double`

Engine2_T — Engine 2 translation
1-by-3 matrix

Engine 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 translation** parameter.

Data Types: `single` | `double`

Engine2_Prop_T — Engine 2 propeller translation
1-by-3 matrix

Engine 2 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 propeller translation** parameter.

Data Types: `single` | `double`

Engine3_T — Engine 3 translation
1-by-3 matrix

Engine 3 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 3 translation** parameter.

Data Types: `single` | `double`

Engine3_Prop_T — Engine 3 propeller translation
1-by-3 matrix

Engine 3 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 3 propeller translation** parameter.

Data Types: `single` | `double`

Engine4_T — Engine 4 translation
1-by-3 matrix

Engine 4 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 4 translation** parameter.

Data Types: `single` | `double`

Engine4_Prop_T — Engine 4 propeller translation
1-by-3 matrix

Engine 4 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 4 propeller translation** parameter.

Data Types: `single` | `double`

Engine5_T — Engine 5 translation
1-by-3 matrix

Engine 5 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 5 translation** parameter.

Data Types: `single` | `double`

Engine5_Prop_T — Engine 5 propeller translation
1-by-3 matrix

Engine 5 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 5 propeller translation** parameter.

Data Types: `single` | `double`

Engine6_T — Engine 6 translation
1-by-3 matrix

Engine 6 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 6 translation** parameter.

Data Types: `single` | `double`

Engine6_Prop_T — Engine 6 propeller translation
1-by-3 matrix

Engine 6 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 6 propeller translation** parameter.

Data Types: `single` | `double`

Engine7_T — Engine 7 translation
1-by-3 matrix

Engine 7 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 7 translation** parameter.

Data Types: `single` | `double`

Engine7_Prop_T — Engine 7 propeller translation
1-by-3 matrix

Engine 7 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 7 propeller translation** parameter.

Data Types: `single` | `double`

Engine8_T — Engine 8 translation
1-by-3 matrix

Engine 8 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 8 translation** parameter.

Data Types: `single` | `double`

Engine8_Prop_T — Engine 8 propeller translation
1-by-3 matrix

Engine 8 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 8 propeller translation** parameter.

Data Types: `single` | `double`

Engine9_T — Engine 9 translation
1-by-3 matrix

Engine 9 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 9 translation** parameter.

Data Types: `single` | `double`

Engine9_Prop_T — Engine 9 propeller translation
1-by-3 matrix

Engine 9 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 9 propeller translation** parameter.

Data Types: `single` | `double`

Engine10_T — Engine 10 translation
1-by-3 matrix

Engine 10 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 10 translation** parameter.

Data Types: `single` | `double`

Engine10_Prop_T — Engine 10 propeller translation
1-by-3 matrix

Engine 10 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 10 propeller translation** parameter.

Data Types: `single` | `double`

Engine11_T — Engine 11 translation
1-by-3 matrix

Engine 11 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 11 translation** parameter.

Data Types: `single` | `double`

Engine11_Prop_T — Engine 11 propeller translation
1-by-3 matrix

Engine 11 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 11 propeller translation** parameter.

Data Types: `single` | `double`

Engine12_T — Engine 12 translation
1-by-3 matrix

Engine 12 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 12 translation** parameter.

Data Types: `single` | `double`

Engine12_Prop_T — Engine 12 propeller translation
1-by-3 matrix

Engine 12 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 12 propeller translation** parameter.

Data Types: `single` | `double`

Engine13_T — Engine 13 translation
1-by-3 matrix

Engine 13 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 13 translation** parameter.

Data Types: `single` | `double`

Engine13_Prop_T — Engine 13 propeller translation
1-by-3 matrix

Engine 13 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 13 propeller translation** parameter.

Data Types: `single` | `double`

Engine14_T — Engine 14 translation
1-by-3 matrix

Engine 14 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 14 translation** parameter.

Data Types: `single` | `double`

Engine14_Prop_T — Engine 14 propeller translation
1-by-3 matrix

Engine 14 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 14 propeller translation** parameter.

Data Types: `single` | `double`

Engine15_T — Engine 15 translation
1-by-3 matrix

Engine 15 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 15 translation** parameter.

Data Types: `single` | `double`

Engine15_Prop_T — Engine 15 propeller translation
1-by-3 matrix

Engine 15 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 15 propeller translation** parameter.

Data Types: `single` | `double`

Engine16_T — Engine 16 translation
1-by-3 matrix

Engine 16 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 16 translation** parameter.

Data Types: `single` | `double`

Engine16_Prop_T — Engine 16 propeller translation
1-by-3 matrix

Engine 16 propeller translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 16 propeller translation** parameter.

Data Types: `single` | `double`

Wing1_T — Wing 1 translation
1-by-3 matrix

Wing 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 translation** parameter.

Data Types: `single` | `double`

Wing1_Left_Flap_T — Wing 1 left flap translation
1-by-3 matrix

Wing 1 left flap translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 left flap translation** parameter.

Data Types: `single` | `double`

Wing1_Right_Flap_T — Wing 1 right flap translation
1-by-3 matrix

Wing 1 right flap translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 right flap translation** parameter.

Data Types: `single` | `double`

Wing1_Left_Aileron_T — Wing 1 left aileron translation
1-by-3 matrix

Wing 1 left aileron translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 left aileron translation** parameter.

Data Types: `single` | `double`

Wing1_Right_Aileron_T — Wing 1 right aileron translation
1-by-3 matrix

Wing 1 right aileron translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 right aileron translation** parameter.

Data Types: `single` | `double`

Wing1_Left_Spoiler_T — Wing 1 left spoiler translation
1-by-3 matrix

Wing 1 left spoiler translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 left spoiler translation** parameter.

Data Types: `single` | `double`

Wing1_Right_Spoiler_T — Wing 1 right spoiler translation
1-by-3 matrix

Wing 1 right spoiler translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 right spoiler translation** parameter.

Data Types: `single` | `double`

Wing2_T — Wing 2 translation
1-by-3 matrix

Wing 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 2 translation** parameter.

Data Types: `single` | `double`

Wing2_Left_Flap_T — Wing 2 left flap translation
1-by-3 matrix

Wing 2 left flap translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 2 left flap translation** parameter.

Data Types: `single` | `double`

Wing2_Right_Flap_T — Wing 2 right flap translation
1-by-3 matrix

Wing 2 right flap translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 2 right flap translation** parameter.

Data Types: `single` | `double`

Horizontal_Stab_T — Horizontal stab translation
1-by-3 matrix

Horizontal stab translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Horizontal stab translation** parameter.

Data Types: `single` | `double`

Left_Elevator_T — Left elevator translation
1-by-3 matrix

Left elevator translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left elevator translation** parameter.

Data Types: `single` | `double`

Right_Elevator_T — Right elevator translation
1-by-3 matrix

Right elevator translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right elevator translation** parameter.

Data Types: `single` | `double`

Left_Rudder_T — Left rudder translation
1-by-3 matrix

Left rudder translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left rudder translation** parameter.

Data Types: `single` | `double`

Right_Rudder_T — Right rudder translation
1-by-3 matrix

Right rudder translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right rudder translation** parameter.

Data Types: `single` | `double`

Nose_Gear_T — Nose gear translation
1-by-3 matrix

Nose gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear translation** parameter.

Data Types: `single` | `double`

Nose_Wheel_T — Nose wheel translation
1-by-3 matrix

Nose wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel translation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door_T — Nose gear door translation
1-by-3 matrix

Nose gear door translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door translation** parameter.

Data Types: `single` | `double`

Left_Gear_T — Left gear translation
1-by-3 matrix

Left gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear translation** parameter.

Data Types: `single` | `double`

Left_Wheel_T — Left wheel translation
1-by-3 matrix

Left wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel translation** parameter.

Data Types: `single` | `double`

Left_Gear_Door_T — Left gear door translation
1-by-3 matrix

Left gear door translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door translation** parameter.

Data Types: `single` | `double`

Right_Gear_T — Right gear translation
1-by-3 matrix

Right gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear translation** parameter.

Data Types: `single` | `double`

Right_Wheel_T — Right wheel translation
1-by-3 matrix

Right wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel translation** parameter.

Data Types: `single` | `double`

Right_Gear_Door_T — Right gear door translation
1-by-3 matrix

Right gear door translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door translation** parameter.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_R — Engine 1 rotation
1-by-3 matrix

Engine 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 rotation** parameter.

Data Types: `single` | `double`

Engine1_Prop_R — Engine 1 propeller rotation
1-by-3 matrix

Engine 1 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 propeller rotation** parameter.

Data Types: `single` | `double`

Engine2_R — Engine 2 rotation
1-by-3 matrix

Engine 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 rotation** parameter.

Data Types: `single` | `double`

Engine2_Prop_R — Engine 2 propeller rotation
1-by-3 matrix

Engine 2 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 propeller rotation** parameter.

Data Types: `single` | `double`

Engine3_R — Engine 3 rotation
1-by-3 matrix

Engine 3 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 3 rotation** parameter.

Data Types: `single` | `double`

Engine3_Prop_R — Engine 3 propeller rotation
1-by-3 matrix

Engine 3 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 3 propeller rotation** parameter.

Data Types: `single` | `double`

Engine4_R — Engine 4 translation
1-by-3 matrix

Engine 4 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 4 translation** parameter.

Data Types: `single` | `double`

Engine4_Prop_R — Engine 4 propeller rotation
1-by-3 matrix

Engine 4 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 4 propeller rotation** parameter.

Data Types: `single` | `double`

Engine5_R — Engine 5 rotation
1-by-3 matrix

Engine 5 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 5 rotation** parameter.

Data Types: `single` | `double`

Engine5_Prop_R — Engine 5 propeller rotation
1-by-3 matrix

Engine 5 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 5 propeller rotation** parameter.

Data Types: `single` | `double`

Engine6_R — Engine 6 rotation
1-by-3 matrix

Engine 6 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 6 rotation** parameter.

Data Types: `single` | `double`

Engine6_Prop_R — Engine 6 propeller rotation
1-by-3 matrix

Engine 6 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 6 propeller rotation** parameter.

Data Types: `single` | `double`

Engine7_R — Engine 7 rotation
1-by-3 matrix

Engine 7 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 7 rotation** parameter.

Data Types: `single` | `double`

Engine7_Prop_R — Engine 7 propeller rotation
1-by-3 matrix

Engine 7 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 7 propeller rotation** parameter.

Data Types: `single` | `double`

Engine8_R — Engine 8 rotation
1-by-3 matrix

Engine 8 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 8 rotation** parameter.

Data Types: `single` | `double`

Engine8_Prop_R — Engine 8 propeller rotation
1-by-3 matrix

Engine 8 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 8 propeller rotation** parameter.

Data Types: `single` | `double`

Engine9_R — Engine 9 rotation
1-by-3 matrix

Engine 9 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 9 rotation** parameter.

Data Types: `single` | `double`

Engine9_Prop_R — Engine 9 propeller rotation
1-by-3 matrix

Engine 9 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 9 propeller rotation** parameter.

Data Types: `single` | `double`

Engine10_R — Engine 10 rotation
1-by-3 matrix

Engine 10 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 10 rotation** parameter.

Data Types: `single` | `double`

Engine10_Prop_R — Engine 10 propeller rotation
1-by-3 matrix

Engine 10 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 10 propeller rotation** parameter.

Data Types: `single` | `double`

Engine11_R — Engine 11 rotation
1-by-3 matrix

Engine 11 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 11 rotation** parameter.

Data Types: `single` | `double`

Engine11_Prop_R — Engine 11 propeller rotation
1-by-3 matrix

Engine 11 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 11 propeller rotation** parameter.

Data Types: `single` | `double`

Engine12_R — Engine 12 rotation
1-by-3 matrix

Engine 12 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 12 rotation** parameter.

Data Types: `single` | `double`

Engine12_Prop_R — Engine 12 propeller rotation
1-by-3 matrix

Engine 12 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 12 propeller rotation** parameter.

Data Types: `single` | `double`

Engine13_R — Engine 13 rotation
1-by-3 matrix

Engine 13 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 13 rotation** parameter.

Data Types: `single` | `double`

Engine13_Prop_R — Engine 13 propeller rotation
1-by-3 matrix

Engine 13 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 13 propeller rotation** parameter.

Data Types: `single` | `double`

Engine14_R — Engine 14 rotation
1-by-3 matrix

Engine 14 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 14 rotation** parameter.

Data Types: `single` | `double`

Engine14_Prop_R — Engine 14 propeller rotation
1-by-3 matrix

Engine 14 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 14 propeller rotation** parameter.

Data Types: `single` | `double`

Engine15_R — Engine 15 rotation
1-by-3 matrix

Engine 15 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 15 rotation** parameter.

Data Types: `single` | `double`

Engine15_Prop_R — Engine 15 propeller rotation
1-by-3 matrix

Engine 15 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 15 propeller rotation** parameter.

Data Types: `single` | `double`

Engine16_R — Engine 16 rotation
1-by-3 matrix

Engine 16 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 16 rotation** parameter.

Data Types: `single` | `double`

Engine16_Prop_R — Engine 16 propeller rotation
1-by-3 matrix

Engine 16 propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 16 propeller rotation** parameter.

Data Types: `single` | `double`

Wing1_R — Wing 1 rotation
1-by-3 matrix

Wing 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 rotation** parameter.

Data Types: `single` | `double`

Wing1_Left_Flap_R — Wing 1 left flap rotation
1-by-3 matrix

Wing 1 left flap rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 left flap rotation** parameter.

Data Types: `single` | `double`

Wing1_Right_Flap_R — Wing 1 right flap rotation
1-by-3 matrix

Wing 1 right flap rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 right flap rotation** parameter.

Data Types: `single` | `double`

Wing1_Left_Aileron_R — Wing 1 left aileron rotation
1-by-3 matrix

Wing 1 left aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 left aileron rotation** parameter.

Data Types: `single` | `double`

Wing1_Right_Aileron_R — Wing 1 right aileron rotation
1-by-3 matrix

Wing 1 right aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 right aileron rotation** parameter.

Data Types: `single` | `double`

Wing1_Left_Spoiler_R — Wing 1 left spoiler rotation
1-by-3 matrix

Wing 1 left spoiler rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 left spoiler rotation** parameter.

Data Types: `single` | `double`

Wing1_Right_Spoiler_R — Wing 1 right spoiler rotation
1-by-3 matrix

Wing 1 right spoiler rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 1 right spoiler rotation** parameter.

Data Types: `single` | `double`

Wing2_R — Wing 2 rotation
1-by-3 matrix

Wing 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 2 rotation** parameter.

Data Types: `single` | `double`

Wing2_Left_Flap_R — Wing 2 left flap rotation
1-by-3 matrix

Wing 2 left flap rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 2 left flap rotation** parameter.

Data Types: `single` | `double`

Wing2_Right_Flap_R — Wing 2 right flap rotation
1-by-3 matrix

Wing 2 right flap rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Wing 2 right flap rotation** parameter.

Data Types: `single` | `double`

Horizontal_Stab_R — Horizontal stab rotation
1-by-3 matrix

Horizontal stab rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Horizontal stab rotation** parameter.

Data Types: `single` | `double`

Left_Elevator_R — Left elevator rotation
1-by-3 matrix

Left elevator rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left elevator rotation** parameter.

Data Types: `single` | `double`

Right_Elevator_R — Right elevator rotation
1-by-3 matrix

Right elevator rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right elevator rotation** parameter.

Data Types: `single` | `double`

Left_Rudder_R — Left rudder rotation
1-by-3 matrix

Left rudder rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left rudder rotation** parameter.

Data Types: `single` | `double`

Right_Rudder_R — Right rudder rotation
1-by-3 matrix

Right rudder rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right rudder rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_R — Nose gear rotation
1-by-3 matrix

Nose gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_R — Nose wheel rotation
1-by-3 matrix

Nose wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door_R — Nose gear door rotation
1-by-3 matrix

Nose gear door rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door rotation** parameter.

Data Types: `single` | `double`

Left_Gear_R — Left gear rotation
1-by-3 matrix

Left gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear rotation** parameter.

Data Types: `single` | `double`

Left_Wheel_R — Left wheel rotation
1-by-3 matrix

Left wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel rotation** parameter.

Data Types: `single` | `double`

Left_Gear_Door_R — Left gear door rotation
1-by-3 matrix

Left gear door rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door rotation** parameter.

Data Types: `single` | `double`

Right_Gear_R — Right gear rotation
1-by-3 matrix

Right gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear rotation** parameter.

Data Types: `single` | `double`

Right_Wheel_R — Right wheel rotation
1-by-3 matrix

Right wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel rotation** parameter.

Data Types: `single` | `double`

Right_Gear_Door_R — Right gear door rotation
1-by-3 matrix

Right gear door rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Aircraft translation
57-by-3 array

Aircraft translation for custom aircraft, returned as an 57-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the aircraft.

The translation applies to these bones of the `Custom` type.

Bone	Index
BODY	1
ENGINE1	2
ENGINE1_PROP	3
ENGINE2	4
ENGINE2_PROP	5
ENGINE3	6
ENGINE3_PROP	7
ENGINE4	8
ENGINE4_PROP	9
ENGINE5	10
ENGINE5_PROP	11
ENGINE6	12
ENGINE6_PROP	13
ENGINE7	14
ENGINE7_PROP	15
ENGINE8	16
ENGINE8_PROP	17
ENGINE9	18
ENGINE9_PROP	19
ENGINE10	20
ENGINE10_PROP	21
ENGINE11	22
ENGINE11_PROP	23
ENGINE12	24
ENGINE12_PROP	25
ENGINE13	26
ENGINE13_PROP	27
ENGINE14	28
ENGINE14_PROP	29
ENGINE15	30

Bone	Index
ENGINE15_PROP	31
ENGINE16	32
ENGINE16_PROP	33
WING1	34
WING1_LEFT_FLAP	35
WING1_RIGHT_FLAP	36
WING1_LEFT_AILERON	37
WING1_RIGHT_AILERON	38
WING1_LEFT_SPOILER	39
WING1_RIGHT_SPOILER	40
WING2	41
WING2_LEFT_FLAP	42
WING2_RIGHT_FLAP	43
HORIZONTAL_STABILIZER	44
LEFT_ELEVATOR	45
RIGHT_ELEVATOR	46
LEFT_RUDDER	47
RIGHT_RUDDER	48
NOSE_GEAR	49
NOSE_WHEEL	50
NOSE_GEAR_DOOR	51
LEFT_GEAR	52
LEFT_WHEEL	53
LEFT_GEAR_DOOR	54
RIGHT_GEAR	55
RIGHT_WHEEL	56
RIGHT_GEAR_DOOR	57

Rotation — Aircraft rotation
57-by-3 array

Aircraft rotation for custom aircraft, returned as an 57-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the aircraft.

Parameters

Propulsion

Engine 1 translation — Option to enable Engine1_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_T** input port.

Programmatic Use
Block Parameter: `Engine1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 1 propeller translation — Option to enable Engine1_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine1_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 translation — Option to enable Engine2_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_T** input port.

Programmatic Use
Block Parameter: `Engine2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 propeller translation — Option to enable Engine2_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine2_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 3 translation — Option to enable Engine3_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine3_T** input port.

Programmatic Use
Block Parameter: `Engine3_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 3 propeller translation — Option to enable Engine3_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine3_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine3_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 4 translation — Option to enable Engine4_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine4_T** input port.

Programmatic Use
Block Parameter: `Engine4_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 4 propeller translation — Option to enable Engine4_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine4_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine4_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 5 translation — Option to enable Engine5_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine5_T** input port.

Programmatic Use
Block Parameter: `Engine5_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 5 propeller translation — Option to enable Engine5_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine5_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine5_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 6 translation — Option to enable Engine6_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine6_T** input port.

Programmatic Use
Block Parameter: `Engine6_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 6 propeller translation — Option to enable Engine6_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine6_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine6_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 7 translation — Option to enable Engine7_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine7_T** input port.

Programmatic Use
Block Parameter: `Engine7_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 7 propeller translation — Option to enable Engine7_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine7_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine7_Prop_T`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 8 translation — Option to enable Engine8_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine8_T** input port.

Programmatic Use
Block Parameter: `Engine8_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 8 propeller translation — Option to enable Engine8_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine8_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine8_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 9 translation — Option to enable Engine9_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine9_T** input port.

Programmatic Use
Block Parameter: `Engine9_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 9 propeller translation — Option to enable Engine9_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine9_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine9_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 10 translation — Option to enable Engine10_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine10_T** input port.

Programmatic Use
Block Parameter: `Engine10_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 10 propeller translation — Option to enable Engine10_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine10_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine10_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 11 translation — Option to enable Engine11_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine11_T** input port.

Programmatic Use
Block Parameter: `Engine11_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 11 propeller translation — Option to enable Engine11_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine11_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine11_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 12 translation — Option to enable Engine12_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine12_T** input port.

Programmatic Use
Block Parameter: `Engine12_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 12 propeller translation — Option to enable Engine12_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine12_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine12_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 13 translation — Option to enable Engine13_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine13_T** input port.

Programmatic Use
Block Parameter: `Engine13_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 13 propeller translation — Option to enable Engine13_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine13_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine13_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 14 translation — Option to enable Engine14_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine14_T** input port.

Programmatic Use
Block Parameter: `Engine14_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 14 propeller translation — Option to enable Engine14_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine14_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine14_Prop_T`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 15 translation — Option to enable Engine15_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine15_T** input port.

Programmatic Use
Block Parameter: `Engine15_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 15 propeller translation — Option to enable Engine15_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine15_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine15_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 16 translation — Option to enable Engine16_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine16_T** input port.

Programmatic Use
Block Parameter: `Engine16_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 16 propeller translation — Option to enable Engine16_Prop_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine16_Prop_T** input port.

Programmatic Use
Block Parameter: `Engine16_Prop_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 1 rotation — Option to enable Engine1_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_R** input port.

Programmatic Use
Block Parameter: `Engine1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 1 propeller rotation — Option to enable Engine1_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine1_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 rotation — Option to enable Engine2_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_R** input port.

Programmatic Use
Block Parameter: `Engine2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 propeller rotation — Option to enable Engine2_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine2_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 3 rotation — Option to enable Engine3_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine3_R** input port.

Programmatic Use
Block Parameter: `Engine3_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 3 propeller rotation — Option to enable Engine3_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine3_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine3_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 4 rotation — Option to enable Engine4_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine4_R** input port.

Programmatic Use
Block Parameter: `Engine4_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 4 propeller rotation — Option to enable Engine4_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine4_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine4_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 5 rotation — Option to enable Engine5_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine5_R** input port.

Programmatic Use
Block Parameter: `Engine5_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 5 propeller rotation — Option to enable Engine5_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine5_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine5_Prop_R`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 6 rotation — Option to enable Engine6_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine6_R** input port.

Programmatic Use
Block Parameter: `Engine6_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 6 propeller rotation — Option to enable Engine6_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine6_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine6_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 7 rotation — Option to enable Engine7_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine7_R** input port.

Programmatic Use
Block Parameter: `Engine7_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 7 propeller rotation — Option to enable Engine7_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine7_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine7_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 8 rotation — Option to enable Engine8_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine8_R** input port.

Programmatic Use
Block Parameter: `Engine8_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 8 propeller rotation — Option to enable Engine8_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine8_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine8_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 9 rotation — Option to enable Engine9_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine9_R** input port.

Programmatic Use
Block Parameter: `Engine9_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 9 propeller rotation — Option to enable Engine9_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine9_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine9_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 10 rotation — Option to enable Engine10_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine10_R** input port.

Programmatic Use
Block Parameter: `Engine10_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 10 propeller rotation — Option to enable Engine10_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine10_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine10_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 11 rotation — Option to enable Engine11_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine11_R** input port.

Programmatic Use
Block Parameter: `Engine11_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 11 propeller rotation — Option to enable Engine11_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine11_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine11_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 12 rotation — Option to enable Engine12_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine12_R** input port.

Programmatic Use
Block Parameter: `Engine12_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 12 propeller rotation — Option to enable Engine12_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine12_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine12_Prop_R`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 13 rotation — Option to enable Engine13_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine13_R** input port.

Programmatic Use
Block Parameter: `Engine13_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 13 propeller rotation — Option to enable Engine13_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine13_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine13_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 14 rotation — Option to enable Engine14_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine14_R** input port.

Programmatic Use
Block Parameter: `Engine14_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 14 propeller rotation — Option to enable Engine14_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine14_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine14_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 15 rotation — Option to enable Engine15_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine15_R** input port.

Programmatic Use
Block Parameter: `Engine15_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 15 propeller rotation — Option to enable Engine15_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine15_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine15_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 16 rotation — Option to enable Engine16_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine16_R** input port.

Programmatic Use
Block Parameter: `Engine16_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 16 propeller rotation — Option to enable Engine16_Prop_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine16_Prop_R** input port.

Programmatic Use
Block Parameter: `Engine16_Prop_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Controls

Wing 1 translation — Option to enable Wing1_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_T** input port.

Programmatic Use
Block Parameter: `Wing1_T`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Wing 1 left flap translation — Option to enable Wing1_Left_Flap_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Left_Flap_T** input port.

Programmatic Use
Block Parameter: `Wing1_Left_Flap_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 right flap translation — Option to enable Wing1_Right_Flap_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Right_Flap_T** input port.

Programmatic Use
Block Parameter: `Wing1_Right_Flap_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 left aileron translation — Option to enable Wing1_Left_Aileron_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Left_Aileron_T** input port.

Programmatic Use
Block Parameter: `Wing1_Left_Aileron_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 right aileron translation — Option to enable Wing1_Right_Aileron_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Right_Aileron_T** input port.

Programmatic Use
Block Parameter: `Wing1_Right_Aileron_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 left spoiler translation — Option to enable Wing1_Left_Spoiler_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Left_Spoiler_T** input port.

Programmatic Use
Block Parameter: `Wing1_Left_Spoiler_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 right spoiler translation — Option to enable Wing1_Right_Spoiler_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Right_Spoiler_T** input port.

Programmatic Use
Block Parameter: `Wing1_Right_Spoiler_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 2 translation — Option to enable Wing2_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing2_T** input port.

Programmatic Use
Block Parameter: `Wing2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 2 left flap translation — Option to enable Wing2_Left_Flap_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing2_Left_Flap_T** input port.

Programmatic Use
Block Parameter: `Wing2_Left_Flap_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 2 right flap translation — Option to enable Wing2_Right_Flap_T input port

`off` (default) | `on`

Select this parameter to enable the **Wing2_Right_Flap_T** input port.

Programmatic Use
Block Parameter: `Wing2_Right_Flap_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Horizontal stab translation — Option to enable Horizontal_Stab_T input port

`off` (default) | `on`

Select this parameter to enable the **Horizontal_Stab_T** input port.

Programmatic Use
Block Parameter: `Horizontal_Stab_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left elevator translation — Option to enable Left_Elevator_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Elevator_T** input port.

Programmatic Use
Block Parameter: `Left_Elevator_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right elevator translation — Option to enable Right_Elevator_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Elevator_T** input port.

Programmatic Use
Block Parameter: `Right_Elevator_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left rudder translation — Option to enable Left_Rudder_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Rudder_T** input port.

Programmatic Use
Block Parameter: `Left_Rudder_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right rudder translation — Option to enable Right_Rudder_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Rudder_T** input port.

Programmatic Use
Block Parameter: `Right_Rudder_T`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Wing 1 rotation — Option to enable Wing1_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_R** input port.

Programmatic Use
Block Parameter: `Wing1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 left flap rotation — Option to enable Wing1_Left_Flap_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Left_Flap_R** input port.

Programmatic Use
Block Parameter: `Wing1_Left_Flap_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 right flap rotation — Option to enable Wing1_Right_Flap_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Right_Flap_R** input port.

Programmatic Use
Block Parameter: `Wing1_Right_Flap_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 left aileron rotation — Option to enable Wing1_Left_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Left_Aileron_R** input port.

Programmatic Use
Block Parameter: `Wing1_Left_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 right aileron rotation — Option to enable Wing1_Right_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Right_Aileron_R** input port.

Programmatic Use
Block Parameter: `Wing1_Right_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 left spoiler rotation — Option to enable Wing1_Left_Spoiler_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Left_Spoiler_R** input port.

Programmatic Use
Block Parameter: `Wing1_Left_Spoiler_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 1 right spoiler rotation — Option to enable Wing1_Right_Spoiler_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing1_Right_Spoiler_R** input port.

Programmatic Use
Block Parameter: `Wing1_Right_Spoiler_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 2 rotation — Option to enable Wing2_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing2_R** input port.

Programmatic Use
Block Parameter: `Wing2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 2 left flap rotation — Option to enable Wing2_Left_Flap_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing2_Left_Flap_R** input port.

Programmatic Use
Block Parameter: `Wing2_Left_Flap_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Wing 2 right flap rotation — Option to enable Wing2_Right_Flap_R input port

`off` (default) | `on`

Select this parameter to enable the **Wing2_Right_Flap_R** input port.

Programmatic Use
Block Parameter: `Wing2_Right_Flap_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Horizontal stab rotation — Option to enable Horizontal_Stab_R input port

`off` (default) | `on`

Select this parameter to enable the **Horizontal_Stab_R** input port.

Programmatic Use
Block Parameter: `Horizontal_Stab_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left elevator rotation — Option to enable Left_Elevator_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Elevator_R** input port.

Programmatic Use
Block Parameter: `Left_Elevator_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right elevator rotation — Option to enable Right_Elevator_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Elevator_R** input port.

Programmatic Use
Block Parameter: `Right_Elevator_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left rudder rotation — Option to enable Left_Rudder_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Rudder_R** input port.

Programmatic Use
Block Parameter: `Left_Rudder_R`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Right rudder rotation — Option to enable Right_Rudder_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Rudder_R** input port.

Programmatic Use
Block Parameter: `Right_Rudder_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Gear

Nose gear translation — Option to enable Nose_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel translation — Option to enable Nose_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_T** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door translation — Option to enable Nose_Gear_Door_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_Door_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear translation — Option to enable Left_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel translation — Option to enable Left_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_T** input port.

Programmatic Use
Block Parameter: `Left_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door translation — Option to enable Left_Gear_Door_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_Door_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear translation — Option to enable Right_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel translation — Option to enable Right_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_T** input port.

Programmatic Use
Block Parameter: `Right_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door translation — Option to enable Right_Gear_Door_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear rotation — Option to enable Nose_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel rotation — Option to enable Nose_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door rotation — Option to enable Nose_Gear_Door_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_Door_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear rotation — Option to enable Left_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_R`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel rotation — Option to enable Left_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_R** input port.

Programmatic Use
Block Parameter: `Left_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door rotation — Option to enable Left_Gear_Door_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_Door_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear rotation — Option to enable Right_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel rotation — Option to enable Right_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_R** input port.

Programmatic Use
Block Parameter: `Right_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door rotation — Option to enable Right_Gear_Door_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_Door_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use
Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History
Introduced in R2023b

R2024a: Simulation 3D Custom Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame
Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D Custom Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also
Simulation 3D Aircraft | Simulation 3D Scene Configuration

Simulation 3D General Aviation Pack

Generate translation and rotation information for general aviation aircraft

Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D General Aviation Pack block creates translation and rotation information for the Simulation 3D Aircraft block with **Type** set to `General aviation`. Use the Simulation 3D General Aviation Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Aircraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_T — Engine translation
1-by-3 matrix

Engine translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine translation** parameter.

Data Types: `single` | `double`

Rudder_T — Rudder translation
1-by-3 matrix

Rudder translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rudder translation** parameter.

Data Types: `single` | `double`

Elevator_T — Elevator translation
1-by-3 matrix

Elevator translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Elevator translation** parameter.

Data Types: `single` | `double`

Left_Aileron_T — Left aileron translation
1-by-3 matrix

Left aileron translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left aileron translation** parameter.

Data Types: `single` | `double`

Right_Aileron_T — Right aileron translation
1-by-3 matrix

Right aileron translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right aileron translation** parameter.

Data Types: `single` | `double`

Flaps_T — Flaps translation
1-by-3 matrix

Flaps translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps translation** parameter.

Data Types: `single` | `double`

Left_Spoiler_T — Left spoiler translation
1-by-3 matrix

Left spoiler translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left spoiler translation** parameter.

Data Types: `single` | `double`

Right_Spoiler_T — Right spoiler translation
1-by-3 matrix

Right spoiler translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right spoiler translation** parameter.

Data Types: `single` | `double`

Nose_Gear_T — Nose gear translation
1-by-3 matrix

Nose gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear translation** parameter.

Data Types: `single` | `double`

Nose_Wheel_T — Nose wheel translation
1-by-3 matrix

Nose wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel translation** parameter.

Data Types: `single` | `double`

Left_Gear_T — Left gear translation
1-by-3 matrix

Left gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear translation** parameter.

Data Types: `single` | `double`

Left_Wheel_T — Left wheel translation
1-by-3 matrix

Left wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel translation** parameter.

Data Types: `single` | `double`

Right_Gear_T — Right gear translation
1-by-3 matrix

Right gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear translation** parameter.

Data Types: `single` | `double`

Right_Wheel_T — Right wheel translation
1-by-3 matrix

Right wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel translation** parameter.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_R — Engine rotation
1-by-3 matrix

Engine rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine rotation** parameter.

Data Types: `single` | `double`

Rudder_R — Rudder rotation
1-by-3 matrix

Rudder rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rudder rotation** parameter.

Data Types: `single` | `double`

Elevator_R — Elevator rotation
1-by-3 matrix

Elevator rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Elevator rotation** parameter.

Data Types: `single` | `double`

Left_Aileron_R — Left aileron rotation
1-by-3 matrix

Left aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left aileron rotation** parameter.

Data Types: `single` | `double`

Right_Aileron_R — Right aileron rotation
1-by-3 matrix

Right aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right aileron rotation** parameter.

Data Types: `single` | `double`

Flaps_R — Flaps rotation
1-by-3 matrix

Flaps rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps rotation** parameter.

Data Types: `single` | `double`

Left_Spoiler_R — Left spoiler rotation
1-by-3 matrix

Left spoiler rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left spoiler rotation** parameter.

Data Types: `single` | `double`

Right_Spoiler_R — Right spoiler rotation
1-by-3 matrix

Right spoiler rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right spoiler rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_R — Nose gear rotation
1-by-3 matrix

Nose gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_R — Nose wheel rotation
1-by-3 matrix

Nose wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel rotation** parameter.

Data Types: `single` | `double`

Left_Gear_R — Left gear rotation
1-by-3 matrix

Left gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear rotation** parameter.

Data Types: `single` | `double`

Left_Wheel_R — Left wheel rotation
1-by-3 matrix

Left wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel rotation** parameter.

Data Types: `single` | `double`

Right_Gear_R — Right gear rotation
1-by-3 matrix

Right gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear rotation** parameter.

Data Types: `single` | `double`

Right_Wheel_R — Right wheel rotation
1-by-3 matrix

Right wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Aircraft translation
15-by-3 array

Aircraft translation for General Aviation aircraft, returned as a 15-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the aircraft.

The translation applies to these bones of the `General Aviation` type.

Bone	Index
BODY	1
ENGINE1	2

Bone	Index
RUDDER	3
ELEVATOR	4
LEFT_AILERON	5
RIGHT_AILERON	6
FLAPS	7
LEFT_SPOILER	8
RIGHT_SPOILER	9
NOSE_GEAR	10
NOSE_WHEEL	11
LEFT_GEAR	12
LEFT_WHEEL	13
RIGHT_GEAR	14
RIGHT_WHEEL	15

Rotation — Aircraft rotation
15-by-3 array

Aircraft rotation for General Aviation aircraft, returned as an 15-by-3 array:

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the aircraft.

Parameters

Propulsion

Engine translation — Option to enable Engine1_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_T** input port.

Programmatic Use
Block Parameter: `Engine1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine rotation — Option to enable Engine1_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_R** input port.

Programmatic Use

Block Parameter: `Engine1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Controls

Rudder translation — Option to enable Rudder_T input port

`off` (default) | `on`

Select this parameter to enable the **Rudder_T** input port.

Programmatic Use

Block Parameter: `Rudder_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Elevator translation — Option to enable Elevator_T input port

`off` (default) | `on`

Select this parameter to enable the **Elevator_T** input port.

Programmatic Use

Block Parameter: `Elevator_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left aileron translation — Option to enable Left_Aileron_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Aileron_T** input port.

Programmatic Use

Block Parameter: `Left_Aileron_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right aileron translation — Option to enable Right_Aileron_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Aileron_T** input port.

Programmatic Use

Block Parameter: `Right_Aileron_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Flaps translation — Option to enable Flaps_T input port

`off` (default) | `on`

Select this parameter to enable the **Flaps_T** input port.

Programmatic Use
Block Parameter: `Flaps_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left spoiler translation — Option to enable Left_Spoiler_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Spoiler_T** input port.

Programmatic Use
Block Parameter: `Left_Spoiler_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right spoiler translation — Option to enable Right_Spoiler_T input port

`off` (default) | `on`

Select this parameter to enable the **Right_Spoiler_T** input port.

Programmatic Use
Block Parameter: `Right_Spoiler_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rudder rotation — Option to enable Rudder_R input port

`off` (default) | `on`

Select this parameter to enable the **Rudder_R** input port.

Programmatic Use
Block Parameter: `Rudder_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Elevator rotation — Option to enable Elevator_R input port

`off` (default) | `on`

Select this parameter to enable the **Elevator_R** input port.

Programmatic Use
Block Parameter: `Elevator_R`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left aileron rotation — Option to enable Left_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Aileron_R** input port.

Programmatic Use
Block Parameter: `Left_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right aileron rotation — Option to enable Right_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Aileron_R** input port.

Programmatic Use
Block Parameter: `Right_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Flaps rotation — Option to enable Flaps_R input port

`off` (default) | `on`

Select this parameter to enable the **Flaps_R** input port.

Programmatic Use
Block Parameter: `Flaps_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left spoiler rotation — Option to enable Left_Spoiler_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Spoiler_R** input port.

Programmatic Use
Block Parameter: `Left_Spoiler_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right spoiler rotation — Option to enable Right_Spoiler_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Spoiler_R** input port.

Programmatic Use
Block Parameter: `Right_Spoiler_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Gear

Nose gear translation — Option to enable Nose_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel translation — Option to enable Nose_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_T** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear translation — Option to enable Left_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel translation — Option to enable Left_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_T** input port.

Programmatic Use
Block Parameter: `Left_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`

Default: 'off'

Right gear translation — Option to enable Right_Gear_T input port

off (default) | on

Select this parameter to enable the **Right_Gear_T** input port.

Programmatic Use
Block Parameter: Right_Gear_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Right wheel translation — Option to enable Right_Wheel_T input port

off (default) | on

Select this parameter to enable the **Right_Wheel_T** input port.

Programmatic Use
Block Parameter: Right_Wheel_T
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Nose gear rotation — Option to enable Nose_Gear_R input port

off (default) | on

Select this parameter to enable the **Nose_Gear_R** input port.

Programmatic Use
Block Parameter: Nose_Gear_R
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Nose wheel rotation — Option to enable Nose_Wheel_R input port

off (default) | on

Select this parameter to enable the **Nose_Wheel_R** input port.

Programmatic Use
Block Parameter: Nose_Wheel_R
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Left gear rotation — Option to enable Left_Gear_R input port

off (default) | on

Select this parameter to enable the **Left_Gear_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel rotation — Option to enable Left_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_R** input port.

Programmatic Use
Block Parameter: `Left_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear rotation — Option to enable Right_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel rotation — Option to enable Right_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_R** input port.

Programmatic Use
Block Parameter: `Right_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use
Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History
Introduced in R2023b

R2024a: Simulation 3D General Aviation Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame
Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D General Aviation Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.

- **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.

- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also

Simulation 3D Aircraft | Simulation 3D Scene Configuration

Simulation 3D Helicopter Pack

Generate translation and rotation information for helicopter

Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Helicopter Pack block creates translation and rotation information for the Simulation 3D Rotorcraft block with **Type** set to `Helicopter`. Use the Simulation 3D Helicopter Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Rotorcraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_T — Engine 1 translation
1-by-3 matrix

Engine 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 translation** parameter.

Data Types: `single` | `double`

Engine2_T — Engine 2 translation
1-by-3 matrix

Engine 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 translation** parameter.

Data Types: `single` | `double`

Rotor1_T — Rotor 1 translation
1-by-3 matrix

Rotor 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 1 translation** parameter.

Data Types: `single` | `double`

Rotor2_T — Rotor 2 translation
1-by-3 matrix

Rotor 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 2 translation** parameter.

Data Types: `single` | `double`

Nose_Gear_T — Nose gear translation
1-by-3 matrix

Nose gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear translation** parameter.

Data Types: `single` | `double`

Nose_Wheel_T — Nose wheel translation
1-by-3 matrix

Nose wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel translation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door1_T — Nose gear door 1 translation
1-by-3 matrix

Nose gear door 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 1 translation** check box.

Data Types: `single` | `double`

Nose_Gear_Door2_T — Nose gear door 2 translation
1-by-3 matrix

Nose gear door 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 2 translation** check box.

Data Types: `single` | `double`

Left_Gear_T — Left gear translation
1-by-3 matrix

Left gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear translation** parameter.

Data Types: `single` | `double`

Left_Wheel_T — Left wheel translation
1-by-3 matrix

Left wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel translation** parameter.

Data Types: `single` | `double`

Left_Gear_Door1_T — Left gear door 1 translation
1-by-3 matrix

Left gear door 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 1 translation** check box.

Data Types: `single` | `double`

Left_Gear_Door2_T — Left gear door 2 translation
1-by-3 matrix

Left gear door 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 2 translation** check box.

Data Types: `single` | `double`

Right_Gear_T — Right gear translation
1-by-3 matrix

Right gear translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear translation** parameter.

Data Types: `single` | `double`

Right_Wheel_T — Right wheel translation
1-by-3 matrix

Right wheel translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel translation** parameter.

Data Types: `single` | `double`

Right_Gear_Door1_T — Right gear door 1 translation
1-by-3 matrix

Right gear door 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 1 translation** check box.

Data Types: `single` | `double`

Right_Gear_Door2_T — Right gear door 2 translation
1-by-3 matrix

Right gear door 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 2 translation** check box.

Data Types: `single` | `double`

Sensor1_T — Sensor 1 translation
1-by-3 matrix

Sensor 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 1 translation** parameter.

Data Types: `single` | `double`

Sensor2_T — Sensor 2 translation
1-by-3 matrix

Sensor 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 2 translation** parameter.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine1_R — Engine 1 rotation
1-by-3 matrix

Engine 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 1 rotation** parameter.

Data Types: `single` | `double`

Engine2_R — Engine 2 rotation
1-by-3 matrix

Engine 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine 2 rotation** parameter.

Data Types: `single` | `double`

Rotor1_R — Rotor 1 rotation
1-by-3 matrix

Rotor 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 1 rotation** parameter.

Data Types: `single` | `double`

Rotor2_R — Rotor 2 rotation
1-by-3 matrix

Rotor 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 2 rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_R — Nose gear rotation
1-by-3 matrix

Nose gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_R — Nose wheel rotation
1-by-3 matrix

Nose wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel rotation** parameter.

Data Types: `single` | `double`

Nose_Gear_Door1_R — Nose gear door 1 rotation
1-by-3 matrix

Nose gear door 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 1 rotation** check box.

Data Types: `single` | `double`

Nose_Gear_Door2_R — Nose gear door 2 rotation
1-by-3 matrix

Nose gear door 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose gear door 2 rotation** check box.

Data Types: `single` | `double`

Left_Gear_R — Left gear rotation
1-by-3 matrix

Left gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear rotation** parameter.

Data Types: `single` | `double`

Left_Wheel_R — Left wheel rotation
1-by-3 matrix

Left wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel rotation** parameter.

Data Types: `single` | `double`

Left_Gear_Door1_R — Left gear door 1 rotation
1-by-3 matrix

Left gear door 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 1 rotation** check box.

Data Types: `single` | `double`

Left_Gear_Door2_R — Left gear door 2 rotation
1-by-3 matrix

Left gear door 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left gear door 2 rotation** check box.

Data Types: `single` | `double`

Right_Gear_R — Right gear rotation
1-by-3 matrix

Right gear rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear rotation** parameter.

Data Types: `single` | `double`

Right_Wheel_R — Right wheel rotation
1-by-3 matrix

Right wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel rotation** parameter.

Data Types: `single` | `double`

Right_Gear_Door1_R — Right gear door 1 rotation
1-by-3 matrix

Right gear door 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 1 rotation** check box.

Data Types: `single` | `double`

Right_Gear_Door2_R — Right gear door 2 rotation
1-by-3 matrix

Right gear door 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right gear door 2 rotation** check box.

Data Types: `single` | `double`

Sensor1_R — Sensor 1 rotation
1-by-3 matrix

Sensor 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 1 rotation** parameter.

Data Types: `single` | `double`

Sensor2_R — Sensor 2 rotation
1-by-3 matrix

Sensor 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 2 rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Helicopter translation
19-by-3 array

Helicopter translation for helicopter rotorcraft, returned as an 19-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the rotorcraft.

The translation applies to these bones of the `Helicopter` type.

Bone	Index
BODY	1
ENGINE1	2
ENGINE2	3
ROTOR1	4
ROTOR2	5
NOSE_GEAR	6
NOSE_WHEEL	7
NOSE_GEAR_DOOR1	8
NOSE_GEAR_DOOR2	9
LEFT_GEAR	10
LEFT_WHEEL	11
LEFT_GEAR_DOOR1	12
LEFT_GEAR_DOOR2	13
RIGHT_GEAR	14
RIGHT_WHEEL	15
RIGHT_GEAR_DOOR1	16
RIGHT_GEAR_DOOR2	17
SENSOR1	18
SENSOR2	19

Rotation — Rotorcraft rotation
19-by-3 array

Rotorcraft rotation for helicopter rotorcraft, returned as an 19-by-3 array.

The rotation applies to the same bones as listed for the "Translation" on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the rotorcraft.

Parameters

Propulsion

Engine 1 translation — Option to enable Engine1_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_T** input port.

Programmatic Use
Block Parameter: `Engine1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 translation — Option to enable Engine2_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_T** input port.

Programmatic Use
Block Parameter: `Engine2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 1 translation — Option to enable Rotor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor1_T** input port.

Programmatic Use
Block Parameter: `Rotor1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 2 translation — Option to enable Rotor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor2_T** input port.

Programmatic Use
Block Parameter: `Rotor2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 1 rotation — Option to enable Engine1_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine1_R** input port.

Programmatic Use
Block Parameter: `Engine1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine 2 rotation — Option to enable Engine2_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine2_R** input port.

Programmatic Use
Block Parameter: `Engine2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 1 rotation — Option to enable Rotor1_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor1_R** input port.

Programmatic Use
Block Parameter: `Rotor1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 2 rotation — Option to enable Rotor2_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor2_R** input port.

Programmatic Use
Block Parameter: `Rotor2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Gear

Nose gear translation — Option to enable Nose_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel translation — Option to enable Nose_Wheel_T input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_T** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 1 translation — Option to enable Nose_Gear_Door1_T input port

`off` (default) | `on`

Select this check box to enable the **Nose_Gear_Door1_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 2 translation — Option to enable Nose_Gear_Door2_T input port

`off` (default) | `on`

Select this check box to enable the **Nose_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear translation — Option to enable Left_Gear_T input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel translation — Option to enable Left_Wheel_T input port

off (default) | on

Select this parameter to enable the **Left_Wheel_T** input port.

Programmatic Use
Block Parameter: `Left_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 1 translation — Option to enable Left_Gear_Door1_T input port

off (default) | on

Select this check box to enable the **Left_Gear_Door1_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 2 translation — Option to enable Left_Gear_Door2_T input port

off (default) | on

Select this check box to enable the **Left_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear translation — Option to enable Right_Gear_T input port

off (default) | on

Select this parameter to enable the **Right_Gear_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel translation — Option to enable Right_Wheel_T input port

off (default) | on

Select this parameter to enable the **Right_Wheel_T** input port.

Programmatic Use
Block Parameter: `Right_Wheel_T`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Right gear door 1 translation — Option to enable Right_Gear_Door1_T input port

`off` (default) | `on`

Select this check box to enable the **Right_Gear_Door1_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 2 translation — Option to enable Right_Gear_Door2_T input port

`off` (default) | `on`

Select this check box to enable the **Right_Gear_Door2_T** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear rotation — Option to enable Nose_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Gear_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel rotation — Option to enable Nose_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 1 rotation — Option to enable Nose_Gear_Door1_R input port

`off` (default) | `on`

Select this check box to enable the **Nose_Gear_Door1_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose gear door 2 rotation — Option to enable Nose_Gear_Door2_R input port

`off` (default) | `on`

Select this check box to enable the **Nose_Gear_Door2_R** input port.

Programmatic Use
Block Parameter: `Nose_Gear_Door2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear rotation — Option to enable Left_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Gear_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left wheel rotation — Option to enable Left_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_R** input port.

Programmatic Use
Block Parameter: `Left_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 1 rotation — Option to enable Left_Gear_Door1_R input port

`off` (default) | `on`

Select this check box to enable the **Left_Gear_Door1_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left gear door 2 rotation — Option to enable Left_Gear_Door2_R input port

`off` (default) | `on`

Select this check box to enable the **Left_Gear_Door2_R** input port.

Programmatic Use
Block Parameter: `Left_Gear_Door2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear rotation — Option to enable Right_Gear_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Gear_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel rotation — Option to enable Right_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_R** input port.

Programmatic Use
Block Parameter: `Right_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 1 rotation — Option to enable Right_Gear_Door1_R input port

`off` (default) | `on`

Select this check box to enable the **Right_Gear_Door1_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right gear door 2 rotation — Option to enable Right_Gear_Door2_R input port

`off` (default) | `on`

Select this check box to enable the **Right_Gear_Door2_R** input port.

Programmatic Use
Block Parameter: `Right_Gear_Door2_R`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Sensors

Sensor 1 translation — Option to enable Sensor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_T** input port.

Programmatic Use
Block Parameter: `Sensor1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 2 translation — Option to enable Sensor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_T** input port.

Programmatic Use
Block Parameter: `Sensor2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 1 rotation — Option to enable Sensor1_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_R** input port.

Programmatic Use
Block Parameter: `Sensor1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 2 rotation — Option to enable Sensor2_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_R** input port.

Programmatic Use
Block Parameter: `Sensor2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use
Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History

Introduced in R2023b

R2024a: Simulation 3D Helicopter Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame

Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D Helicopter Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

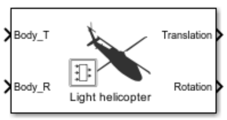
- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also

Simulation 3D Rotorcraft | Simulation 3D Scene Configuration

Simulation 3D Light Helicopter Pack

Generate translation and rotation information for light helicopter



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Light Helicopter Pack block creates translation and rotation information for the Simulation 3D Rotorcraft block with **Type** set to `Light helicopter`. Use the Simulation 3D Light Helicopter Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Rotorcraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine_T — Engine translation
1-by-3 matrix

Engine translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine translation** parameter.

Data Types: `single` | `double`

Rotor1_T — Rotor 1 translation
1-by-3 matrix

Rotor 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 1 translation** parameter.

Data Types: `single` | `double`

Rotor2_T — Rotor 2 translation
1-by-3 matrix

Rotor 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 2 translation** parameter.

Data Types: `single` | `double`

Sensor1_T — Sensor 1 translation
1-by-3 matrix

Sensor 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 1 translation** parameter.

Data Types: `single` | `double`

Sensor2_T — Sensor 2 translation
1-by-3 matrix

Sensor 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 2 translation** parameter.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Engine_R — Engine rotation
1-by-3 matrix

Engine rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Engine rotation** parameter.

Data Types: `single` | `double`

Rotor1_R — Rotor 1 rotation
1-by-3 matrix

Rotor 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 1 rotation** parameter.

Data Types: `single` | `double`

Rotor2_R — Rotor 2 rotation
1-by-3 matrix

Rotor 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 2 rotation** parameter.

Data Types: `single` | `double`

Sensor1_R — Sensor 1 rotation
1-by-3 matrix

Sensor 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 1 rotation** parameter.

Data Types: `single` | `double`

Sensor2_R — Sensor 2 rotation
1-by-3 matrix

Sensor 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 2 rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Light helicopter translation
6-by-3 array

Rotorcraft translation for light helicopter, returned as a 6-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the rotorcraft.

The translation applies to these bones of the `Light helicopter` type.

Bone	Index
BODY	1
ENGINE	2
ROTOR1	3
ROTOR2	4
SENSOR1	5
SENSOR2	6

Rotation — Rotorcraft rotation
6-by-3 array

Rotorcraft rotation for light helicopter, returned as a 6-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the rotorcraft.

Parameters

Propulsion

Engine translation — Option to enable Engine_T input port

`off` (default) | `on`

Select this parameter to enable the **Engine_T** input port.

Programmatic Use
Block Parameter: `Engine_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 1 translation — Option to enable Rotor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor1_T** input port.

Programmatic Use
Block Parameter: `Rotor1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 2 translation — Option to enable Rotor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor2_T** input port.

Programmatic Use
Block Parameter: `Rotor2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Engine rotation — Option to enable Engine_R input port

`off` (default) | `on`

Select this parameter to enable the **Engine_R** input port.

Programmatic Use
Block Parameter: `Engine_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 1 rotation — Option to enable Rotor1_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor1_R** input port.

Programmatic Use
Block Parameter: `Rotor1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 2 rotation — Option to enable Rotor2_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor2_R** input port.

Programmatic Use
Block Parameter: `Rotor2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensors

Sensor 1 translation — Option to enable Sensor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_T** input port.

Programmatic Use
Block Parameter: `Sensor1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 2 translation — Option to enable Sensor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_T** input port.

Programmatic Use
Block Parameter: `Sensor2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 1 rotation — Option to enable Sensor1_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_R** input port.

Programmatic Use
Block Parameter: `Sensor1_R`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Sensor 2 rotation — Option to enable Sensor2_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_R** input port.

Programmatic Use
Block Parameter: `Sensor2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use

Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History

Introduced in R2023b

R2024a: Simulation 3D Light Helicopter Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame

Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D Light Helicopter Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

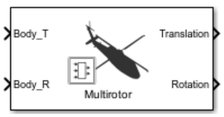
- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also

Simulation 3D Rotorcraft | Simulation 3D Scene Configuration

Simulation 3D Multirotor Pack

Generate translation and rotation information for multirotor rotorcraft



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Multirotor Pack block creates translation and rotation information for the Simulation 3D Rotorcraft block with **Type** set to `Multirotor`. Use the Simulation 3D Multirotor Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Rotorcraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Rotor1_T — Rotor 1 translation
1-by-3 matrix

Rotor 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 1 translation** parameter.

Data Types: `single` | `double`

Rotor2_T — Rotor 2 translation
1-by-3 matrix

Rotor 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 2 translation** parameter.

Data Types: `single` | `double`

Rotor3_T — Rotor 3 translation
1-by-3 matrix

Rotor 3 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 3 translation** parameter.

Data Types: `single` | `double`

Rotor4_T — Rotor 4 translation
1-by-3 matrix

Rotor 4 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 4 translation** parameter.

Data Types: `single` | `double`

Rotor5_T — Rotor 5 translation
1-by-3 matrix

Rotor 5 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 5 translation** parameter.

Data Types: `single` | `double`

Rotor6_T — Rotor 6 translation
1-by-3 matrix

Rotor 6 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 6 translation** parameter.

Data Types: `single` | `double`

Rotor7_T — Rotor 7 translation
1-by-3 matrix

Rotor 7 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 7 translation** parameter.

Data Types: `single` | `double`

Rotor8_T — Rotor 8 translation
1-by-3 matrix

Rotor 8 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 8 translation** parameter.

Data Types: `single` | `double`

Sensor1_T — Sensor 1 translation
1-by-3 matrix

Sensor 1 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 1 translation** parameter.

Data Types: `single` | `double`

Sensor2_T — Sensor 2 translation
1-by-3 matrix

Sensor 2 translation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 2 translation** parameter.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Rotor1_R — Rotor 1 rotation
1-by-3 matrix

Rotor 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 1 rotation** parameter.

Data Types: `single` | `double`

Rotor2_R — Rotor 2 rotation
1-by-3 matrix

Rotor 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 2 rotation** parameter.

Data Types: `single` | `double`

Rotor3_R — Rotor 3 rotation
1-by-3 matrix

Rotor 3 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 3 rotation** parameter.

Data Types: `single` | `double`

Rotor4_R — Rotor 4 rotation
1-by-3 matrix

Rotor 4 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 4 rotation** parameter.

Data Types: `single` | `double`

Rotor5_R — Rotor 5 rotation
1-by-3 matrix

Rotor 5 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 5 rotation** parameter.

Data Types: `single` | `double`

Rotor6_R — Rotor 6 rotation
1-by-3 matrix

Rotor 6 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 6 rotation** parameter.

Data Types: `single` | `double`

Rotor7_R — Rotor 7 rotation
1-by-3 matrix

Rotor 7 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 7 rotation** parameter.

Data Types: `single` | `double`

Rotor8_R — Rotor 8 rotation
1-by-3 matrix

Rotor 8 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rotor 8 rotation** parameter.

Data Types: `single` | `double`

Sensor1_R — Sensor 1 rotation
1-by-3 matrix

Sensor 1 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 1 rotation** parameter.

Data Types: `single` | `double`

Sensor2_R — Sensor 2 rotation
1-by-3 matrix

Sensor 2 rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Sensor 2 rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Multirotor translation
11-by-3 array

Rotorcraft translation for multirotor, returned as an `11`-by-`3` array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the rotorcraft.

The translation applies to these bones of the `Multirotor` type.

Bone	Index
BODY	1
ROTOR1	2
ROTOR2	3
ROTOR3	4
ROTOR4	5
ROTOR5	6
ROTOR6	7
ROTOR7	8
ROTOR8	9
SENSOR1	10
SENSOR2	11

Rotation — Rotorcraft rotation
11-by-3 array

Rotorcraft rotation for multirotor, returned as an `11`-by-`3` array:

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the rotorcraft.

Parameters

Propulsion

Rotor 1 translation — Option to enable Rotor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor1_T** input port.

Programmatic Use
Block Parameter: `Rotor1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 2 translation — Option to enable Rotor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor2_T** input port.

Programmatic Use
Block Parameter: `Rotor2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 3 translation — Option to enable Rotor3_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor3_T** input port.

Programmatic Use
Block Parameter: `Rotor3_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 4 translation — Option to enable Rotor4_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor4_T** input port.

Programmatic Use
Block Parameter: `Rotor4_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 5 translation — Option to enable Rotor5_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor5_T** input port.

Programmatic Use
Block Parameter: `Rotor5_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 6 translation — Option to enable Rotor6_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor6_T** input port.

Programmatic Use
Block Parameter: `Rotor6_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 7 translation — Option to enable Rotor7_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor7_T** input port.

Programmatic Use
Block Parameter: `Rotor7_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 8 translation — Option to enable Rotor8_T input port

`off` (default) | `on`

Select this parameter to enable the **Rotor8_T** input port.

Programmatic Use
Block Parameter: `Rotor8_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 1 rotation — Option to enable Rotor1_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor1_R** input port.

Programmatic Use
Block Parameter: `Rotor1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 2 rotation — Option to enable Rotor2_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor2_R** input port.

Programmatic Use
Block Parameter: `Rotor2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 3 rotation — Option to enable Rotor3_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor3_R** input port.

Programmatic Use
Block Parameter: `Rotor3_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 4 rotation — Option to enable Rotor4_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor4_R** input port.

Programmatic Use
Block Parameter: `Rotor4_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 5 rotation — Option to enable Rotor5_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor5_R** input port.

Programmatic Use
Block Parameter: `Rotor5_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 6 rotation — Option to enable Rotor6_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor6_R** input port.

Programmatic Use
Block Parameter: `Rotor6_R`

Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 7 rotation — Option to enable Rotor7_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor7_R** input port.

Programmatic Use
Block Parameter: `Rotor7_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Rotor 8 rotation — Option to enable Rotor8_R input port

`off` (default) | `on`

Select this parameter to enable the **Rotor8_R** input port.

Programmatic Use
Block Parameter: `Rotor8_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensors

Sensor 1 translation — Option to enable Sensor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_T** input port.

Programmatic Use
Block Parameter: `Sensor1_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 2 translation — Option to enable Sensor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_T** input port.

Programmatic Use
Block Parameter: `Sensor2_T`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 1 rotation — Option to enable Sensor1_R input port

off (default) | on

Select this parameter to enable the **Sensor1_R** input port.

Programmatic Use
Block Parameter: `Sensor1_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Sensor 2 rotation — Option to enable Sensor2_R input port

off (default) | on

Select this parameter to enable the **Sensor2_R** input port.

Programmatic Use
Block Parameter: `Sensor2_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

NED (default) | ECEF

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

off (default) | on

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use

Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History

Introduced in R2023b

R2024a: Simulation 3D Multirotor Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame

Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D Multirotor Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

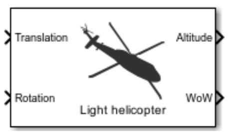
- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also

Simulation 3D Rotorcraft | Simulation 3D Scene Configuration

Simulation 3D Rotorcraft

Implement 3D Rotorcraft



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

Note Simulating models with the Simulation 3D Rotorcraft block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Rotorcraft to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Rotorcraft block implements a rotorcraft in a 3D visualization environment using translation and rotation to place the rotorcraft.

To use this block, your model must contain a Simulation 3D Scene Configuration block. If you set the **Sample time** parameter of this block to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

The block input uses the rotorcraft north-east-down (NED) *right-handed* (RH) *Cartesian* coordinate system.

- *X*-axis — Along rotorcraft longitudinal axis, points forward
- *Y*-axis — Along rotorcraft lateral axis, points to the right
- *Z*-axis — Points downward

For more information, see “About Aerospace Coordinate Systems” on page 2-8.

Tip Verify that the Simulation 3D Rotorcraft block executes before the Simulation 3D Scene Configuration block. That way, Simulation 3D Rotorcraft prepares the signal data before the Unreal Engine 3D visualization environment receives the data. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Rotorcraft — `-1`

For more information about execution order, see “Control and Display Execution Order”.

Skeletons, Bones, and Meshes

Unreal uses a skeleton, bones, and mesh to define a 3D model. A skeleton is comprised of a set of bones. A mesh is the outer covering of the skeleton. Rotorcraft parts are sections of the mesh, such as rotor blades or wheels, which are linked to the bones. For more information, see https://docs.unrealengine.com/4.27/AnimatingObjects/SkeletalMeshAnimation/Skeleton/.

For more information on how the Simulation 3D Rotorcraft block translation input arrays connect to rotorcraft types, see “Algorithms” on page 5-1054.

Ports

Input

Translation — Rotorcraft translation
6-by-3 array | 19-by-3 array | 11-by-3 array

Rotorcraft translation, specified as:

- 6-by-3 array — Rotorcraft **Type** is `Light helicopter`.
- 19-by-3 array — Rotorcraft **Type** is `Helicopter`.
- 11-by-3 array — Rotorcraft **Type** is `Multirotor`.

The signal contains translation [X, Y, Z], in meters, with one row of the array for each bone of the rotorcraft.

The translation applies to these bones of the `Light helicopter` type.

Bone	Index
BODY	1
ENGINE	2
ROTOR1	3
ROTOR2	4
SENSOR1	5
SENSOR2	6

The translation applies to these bones of the `Helicopter` type.

Bone	Index
BODY	1
ENGINE1	2
ENGINE2	3
ROTOR1	4
ROTOR2	5
NOSE_GEAR	6
NOSE_WHEEL	7
NOSE_GEAR_DOOR1	8
NOSE_GEAR_DOOR2	9
LEFT_GEAR	10

Bone	Index
LEFT_WHEEL	11
LEFT_GEAR_DOOR1	12
LEFT_GEAR_DOOR2	13
RIGHT_GEAR	14
RIGHT_WHEEL	15
RIGHT_GEAR_DOOR1	16
RIGHT_GEAR_DOOR2	17
SENSOR1	18
SENSOR2	19

The translation applies to these bones of the `Multirotor` type.

Bone	Index
BODY	1
ROTOR1	2
ROTOR2	3
ROTOR3	4
ROTOR4	5
ROTOR5	6
ROTOR6	7
ROTOR7	8
ROTOR8	9
SENSOR1	10
SENSOR2	11

Data Types: `single` | `double`

Rotation — Rotorcraft and wheel rotation
6-by-3 array | 19-by-3 array | 11-by-3 array

Rotorcraft rotation, specified as:

- 6-by-3 array — Rotorcraft **Type** is `Light helicopter`.
- 19-by-3 array — Rotorcraft **Type** is `Helicopter`.
- 11-by-3 array — Rotorcraft **Type** is `Multirotor`.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the rotorcraft.

Data Types: `single` | `double`

LightStates — Rotorcraft light control
1-by-7 vector of double values

Rotorcraft light control, specified as a 1-by-7 vector of double values. Each element of the vector turns on (1) or off (0) a specific rotorcraft light group. The vector has this order:

- LANDING_LIGHTS
- NOSE_LIGHTS
- ANTICOLLISION_BEACONS
- N/A (Ignored)
- STROBE_LIGHTS
- NAVIGATION_LIGHTS
- POSITION_LIGHTS

Dependencies

To enable this port, set the **Light Configuration** parameter to Configurable lights.

Data Types: single | double

Output

Altitude — Rotorcraft attitude
1-by-5 vector

Rotorcraft attitude, returned as a 1-by-5 vector. The altitudes are, in order:

Light Helicopter	Helicopter	Multirotor
rotorcraft_body	rotorcraft_body	rotorcraft_body
rotorcraft_contact_location1	rotorcraft_front_tire	rotorcraft_contact_location1
rotorcraft_contact_location2	rotorcraft_left_tire	rotorcraft_contact_location2
rotorcraft_contact_location3	rotorcraft_right_tire	rotorcraft_contact_location3
rotorcraft_contact_location4	N/A (0)	rotorcraft_contact_location4

Dependencies

To enable this port, select the **Enable altitude and Wow sensors** check box.

Data Types: double

WoW — Weight on wheels
true | false

Rotorcraft weight on wheels, WoW, logical switch, returned as true if either of the contact locations or main gear tires (left or right) are on the ground. Otherwise, the output is false.

Dependencies

To enable this port, select the **Enable altitude and Wow sensors** check box.

Data Types: Boolean

Parameters

Sample time — Sample time

`-1` (default) | real scalar

Sample time, T_s. The graphics frame rate is the inverse of the sample time.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: real scalar
Default: `'-1'`

Rotorcraft

Type — Rotorcraft type

`Light helicopter` (default) | `Helicopter` | `Multirotor`

Rotorcraft type, specified as `Light helicopter`, `Helicopter`, or `Multirotor`.

Dependencies

To set this parameter, set the **Initial translation** and **Initial rotation** parameters to the matching array size. Failure to appropriately set these array sizes causes an error.

Programmatic Use
Block Parameter: `Mesh`
Type: character vector
Values: `'Light helicopter'` | `'Helicopter'` | `'Multiroter'`
Default: `'Light helicopter'`

Path to light helicopter mesh — Path to light helicopter mesh

`'/MathWorksAerospaceContent/Vehicles/Rotorcraft/LightHelicopter/Mesh/SK_LightHelicopter.SK_LightHelicopter'` (default) | character vector

Path to light helicopter mesh, specified as a character vector.

Dependencies

To enable this parameter, set **Type** to `Light helicopter`.

Programmatic Use
Block Parameter: `MeshPathLight`
Type: character vector
Values: `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/LightHelicopter/Mesh/SK_LightHelicopter.SK_LightHelicopter'`
Default: `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/LightHelicopter/Mesh/SK_LightHelicopter.SK_LightHelicopter'`

Path to helicopter mesh — Path to helicopter mesh

`'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Helicopter/Mesh/SK_Helicopter.SK_Helicopter'` (default) | character vector

Path to helicopter mesh, specified as a character vector.

Dependencies

To enable this parameter, set **Type** to `Helicopter`.

Programmatic Use
Block Parameter: `MeshPath`
Type: character vector
Values: `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Helicopter/Mesh/SK_Helicopter.SK_Helicopter'`
Default: `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Helicopter/Mesh/SK_Helicopter.SK_Helicopter'`

Path to multirotor mesh — Path to multirotor mesh

`'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Multirotor/Mesh/SK_Quadcopter.SK_Quadcopter'` (default) | `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Multirotor/Mesh/SK_QuadcopterMini.SK_QuadcopterMini'` | character vector

Path to multirotor mesh, specified as a character vector.

Dependencies

To enable this parameter, set **Type** to `Multirotor`.

Programmatic Use
Block Parameter: `MeshPathMulti`
Type: character vector
Values: `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Multirotor/Mesh/SK_Quadcopter.SK_Quadcopter'` | `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Multirotor/Mesh/SK_QuadcopterMini.SK_QuadcopterMini'`
Default: `'/MathWorksAerospaceContent/Vehicles/Rotorcraft/Multirotor/Mesh/SK_Quadcopter.SK_Quadcopter'`

Color — Rotorcraft color

`Red` (default) | `Orange` | `Yellow` | `Green` | `Cyan` | `Blue` | `Black` | `White` | `Silver` | `Metal`

Rotorcraft color, specified as `Red`, `Orange`, `Yellow`, `Green`, `Cyan`, `Blue`, `Black`, `White`, `Silver`, or `Metal`.

Programmatic Use
Block Parameter: `AircraftColor`
Type: character vector
Values: `'Red'` | `'Orange'` | `'Yellow'` | `'Green'` | `'Cyan'` | `'Blue'` | `'Black'` | `'White'` | `'Silver'` | `'Metal'`
Default: `'Red'`

Name — Rotorcraft name

`SimulinkVehicle1` (default) | character vector

Rotorcraft name, specified as a character vector. By default, when you use the block in your model, the block sets the **Name** parameter to `SimulinkVehicle`*X*. The value of *X* depends on the number of other simulation 3D blocks that you have in your model.

Programmatic Use
Block Parameter: `ActorName`
Type: character vector
Values: scalar
Default: `'SimulinkVehicle1'`

Initial translation (m) — Initial translation of rotorcraft

`zeros(6,3)` (default) | 19-by-3 array | 11-by-3 array

Initial translation of rotorcraft, specified as an 6-by-3, 19-by-3, or 11-by-3 array.

Dependencies

This parameter must match the rotorcraft type you set in **Type**. Failure to appropriately set these array sizes causes an error.

Programmatic Use
Block Parameter: `Translation`
Type: character vector
Values: 6-by-3 array | 19-by-3 array | 11-by-3 array
Default: `'zeros(6,3)'`

Data Types: `single` | `double`

Initial rotation (rad) — Rotorcraft rotation

`zeros(6,3)` (default) | 19-by-3 array | 11-by-3 array

Initial rotation of rotorcraft, specified as an 6-by-3, 19-by-3, or 11-by-3 array.

Dependencies

This parameter must match the rotorcraft type you set in **Type**. Failure to appropriately set these array sizes causes an error.

Programmatic Use
Block Parameter: `Rotation`
Type: character vector
Values: 6-by-3 array | 19-by-3 array | 11-by-3 array
Default: `'zeros(6,3)'`

Data Types: `single` | `double`

Altitude Sensor

Enable altitude and WoW sensors — Altitude and WoW sensors

on (default) | off

To enable output ports for altitude and WoW sensors, select this check box. Otherwise, clear this check box. For more information on the altitude and WoW sensors, see "Altitude Sensors" on page 5-1054 and "WoW Sensors" on page 5-1055.

Programmatic Use
Block Parameter: `IsGHSensorEnabled`
Type: character vector

Values: `'on'` | `'off'`
Default: `'on'`

Show sensor rays in viewer — Option to show sensor rays

off (default) | on

To show sensor rays in the viewer, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `AreGHRaysVisible`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Interpret vector outputs as 1-D — Control reshaping on sensor output ports

off (default) | on

Control if reshaping occurs on the **Altitude WoW** sensor output ports:

- Select this check box to enable reshaping.
- Clear this check box to disable reshaping.

Programmatic Use

Block Parameter: `IsOutputReshapeEnabled`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Length of rays (m) — Length of rays

`1524` (default) | real scalar

Length of rays, specified as a real scalar, in meters. The length of the rays limits the altitude detection. For example, if the vertical distance to the ground beneath the rotorcraft origin is greater than the length of the rays plus the rotorcraft body *Z* offset, the altitude sensor returns `-1` for the first value.

Dependencies

To enable this parameter, select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `GHRayLength`
Type: character vector
Values: real scalar
Default: `'1524'`

Rotorcraft body Z offset (m) — Rotorcraft body *Z* offset

`1.169` (default) | real scalar

Rotorcraft body *Z* offset, specified as a real scalar, in meters.

Dependencies

To enable this parameter, select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `GHBodyOffset`
Type: character vector
Values: real scalar
Default: `'1.169'`

Ground-contact location 1 (m) — First ground contact location

`[1.47 -1.04 1.169]` (default) | 3-element vector

First ground contact location, specified as a 3-element vector, in meters.

Dependencies

To enable this parameter:

- Set **Type** to `Light helicopter` or `Multirotor`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `GHContactLocation1`
Type: character vector
Values: 3-element vector
Default: `'[1.47 -1.04 1.169]'`

Ground-contact location 2 (m) — Second ground contact location

`[-1.24 -1.04 1.169]` (default) | 3-element vector

Second ground contact location, specified as a 3-element vector, in meters.

Dependencies

To enable this parameter:

- Set **Type** to `Light helicopter` or `Multirotor`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `GHContactLocation2`
Type: character vector
Values: 3-element vector
Default: `'[-1.24 -1.04 1.169]'`

Ground-contact location 3 (m) — Third ground contact location

`[-1.24 1.04 1.169]` | 3-element vector

Third ground contact location, specified as a 3-element vector, in meters.

Dependencies

To enable this parameter:

- Set **Type** to `Light helicopter` or `Multirotor`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use
Block Parameter: `GHContactLocation3`
Type: character vector
Values: 3-element vector
Default: `'[-1.24 1.04 1.169]'`

Ground-contact location 4 (m) — Fourth ground contact location

`[1.47 1.04 1.169]` | 3-element vector

Fourth ground contact location, specified as a 3-element vector, in meters.

Dependencies

To enable this parameter:

- Set **Type** to `Light helicopter` or `Multirotor`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use
Block Parameter: `GHContactLocation4`
Type: character vector
Values: 3-element vector
Default: `'[1.47 1.04 1.169]'`

Front tire radius (m) — Front gear tire radius

`0.196` (default) | real scalar

Front gear tire radius, specified as a real scalar, in meters. The front gear altitude ray originates at the front gear axle center plus the front gear tire radius *Z* offset.

Dependencies

To enable this parameter:

- Set **Type** to `Helicopter`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use
Block Parameter: `GHFrontTireRadius`
Type: character vector
Values: real scalar
Default: `'0.196'`

Left tire radius (m) — Left gear tire radius

`0.203` (default) | real scalar

Left gear tire radius, specified as a real scalar, in meters.

Dependencies

To enable this parameter:

- Set **Type** to `Helicopter`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `GHLeftTireRadius`
Type: character vector
Values: real scalar
Default: `'0.203'`

Right tire radius (m) — Right gear tire radius

`0.203` (default) | real scalar

Right gear tire radius, specified as a real scalar, in meters.

Dependencies

To enable this parameter:

- Set **Type** to `Helicopter`.
- Select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `GHRightTireRadius`
Type: character vector
Values: real scalar
Default: `'0.203'`

WoW sensor tolerance (m) — Wow sensor tolerance

`0.05` (default) | real scalar

Wow sensor tolerance, specified as a real scalar.

Dependencies

To enable this parameter, select the **Enable altitude and Wow sensors** check box.

Programmatic Use

Block Parameter: `WoWSensorTolerance`
Type: character vector
Values: real scalar
Default: `'0.05'`

Light Configuration

Light configuration — Light configurations options

`Automatic lights` (default) | `Configurable lights` | `Lights off`

Light configuration options, specified as one of these values:

- `Automatic lights` — Use default rotorcraft lighting configuration that provides realistic pattern cycling.
- `Configurable lights` — Configure rotorcraft lighting parameters.
- `Lights off` — Turn off all rotorcraft lights.

Dependencies

- Setting this parameter to `Automatic lights` disables the configurability of all other **Light Configuration** parameters. The block uses the default parameter values for rotorcraft lighting values.
- Setting this parameter to `Configurable lights` enables the configurability of the **Light Configuration** parameters according to the rotorcraft type.
- Setting **Type** to a given rotorcraft type and **Light Configuration** to `Configurable lights` enables the configurability of the lighting parameters in use for each rotorcraft.
- Setting this parameter to `Lights off` disables the configurability of all other **Light Configuration** parameters. The block turns off all rotorcraft lighting.

Programmatic Use

Block Parameter: `LightsConfig`
Type: character vector
Values: `'Automatic lights'` | `'Configurable lights'` | `'Lights off'`
Default: `'Automatic lights'`

Landing light intensity (cd) — Landing lights intensity

`300000` (default) | positive scalar

Landing light intensity, specified as a positive scalar, in candela.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `LandingLightIntensity`
Type: character vector
Values: positive scalar
Default: `'300000'`

Landing light cone half angle (deg) — Landing lights cone half angle

`15` (default) | positive scalar

Landing lights cone half angle, specified as a positive scalar, in degrees.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use

Block Parameter: `LandingLightConeAngle`
Type: character vector
Values: positive scalar
Default: `'15'`

Landing light color [r g b] — Landing light color

`[1 1 1]` (default) | 3-element vector

Landing light color, specified as a 3-element vector [red green blue].

Dependencies

To enable this parameter:

- Set **Type** to **Multirotor**.
- Set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `LandingLightColor`
Type: character vector
Values: 3-element vector
Default: `'[1 1 1]'`

Landing light location — Landing light location

`[0 0 0]` (default) | 3-element vector

Landing light location with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `HelicopterLeftLandingLightLocation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Landing light orientation (deg) — Landing light orientation

`[0 0 0]` (default) | 3-element vector

Landing light orientation with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `HelicopterLeftLandingLightOrientation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Nose light intensity (cd) — Nose lights intensity

`150000` (default) | positive scalar

Nose lights intensity, specified as a positive scalar, in candela.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `NoseLightIntensity`
Type: character vector
Values: positive scalar
Default: `'150000'`

Nose light cone half angle (deg) — Nose light cone half angle

`36` (default) | positive scalar

Nose light cone half angle, specified as a positive scalar, in degrees.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `NoseLightConeAngle`
Type: character vector
Values: positive scalar
Default: `'36'`

Nose light location — Nose light location

`[0 0 0]` (default) | 3-element vector

Nose light location with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `HelicopterNoseLightLocation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Nose light orientation (deg) — Nose light orientation

`[0 0 0]` (default) | 3-element vector

Nose light orientation with respect to the associated bone of the skeletal mesh, specified as a 3-element vector.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `HelicopterNoseLightOrientation`
Type: character vector
Values: 3-element vector
Default: `'[0 0 0]'`

Navigation lights intensity — Navigation lights intensity

`500` (default) | positive scalar

Navigation lights intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `NavLightIntensity`
Type: character vector
Values: positive scalar
Default: `'500'`

Position light intensity — Position light intensity

`500` (default) | positive scalar

Position light intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `PositionLightIntensity`
Type: character vector
Values: positive scalar
Default: `'500'`

Strobe light intensity — Strobe light intensity

`5000` (default) | positive scalar

Strobe light intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `StrobeLightIntensity`
Type: character vector
Values: positive scalar
Default: `'5000'`

Strobe color [r g b] — Strobe color

`[1 1 1]` (default) | 3-element vector

Strobe light color, specified as a 3-element vector [red green blue].

Dependencies

To enable this parameter:

- Set **Type** to `Multirotor`.
- Set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `StrobeLightColor`
Type: character vector
Values: 3-element vector
Default: `'[1 1 1]'`

Strobe period (s) — Strobe period

`1.5` (default) | positive scalar

Strobe period, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `StrobePeriod`
Type: character vector
Values: positive scalar
Default: `'1.5'`

Strobe pulse width (% of period) — Strobe pulse width

`6` (default) | positive scalar

Strobe pulse width, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `StrobePulseWidth`
Type: character vector
Values: positive scalar
Default: `'6'`

Beacon light intensity — Beacon light intensity

`4000` (default) | positive scalar

Beacon light intensity, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconLightIntensity`
Type: character vector
Values: positive scalar
Default: `'4000'`

Beacon color [r g b] — Beacon color

`[1 0 0]` (default) | 3-element vector

Beacon color, specified as a 3-element vector [red green blue].

Dependencies

To enable this parameter:

- Set **Type** to **Multirotor**.
- Set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconLightColor`
Type: character vector
Values: 3-element vector
Default: `'[1 0 0]'`

Beacon period (s) — Beacon period

`1.5` (default) | positive scalar

Beacon period, specified as a positive scalar, in seconds.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconPeriod`
Type: character vector
Values: positive scalar
Default: `'1.5'`

Beacon pulse width (% of period) — Beacon pulse width

`10` (default) | positive scalar

Beacon pulse width, specified as a positive scalar.

Dependencies

To enable this parameter, set **Light configuration** to `Configurable lights`.

Programmatic Use
Block Parameter: `BeaconPulseWidth`
Type: character vector
Values: positive scalar
Default: `'10'`

Algorithms

All Rotorcraft Type Bones

For all rotorcraft types, the Simulation 3D Rotorcraft block enables six degrees of freedom (6DOF) for all rotorcraft bones. The default poses for each bone in the skeletal mesh affects how the mesh deforms when you manipulate each degree of freedom.

Altitude Sensors

Altitude sensors return the vertical distance to the ground below a vehicle origin, mimicking a radar altimeter. They also return below ground contact points, used by “WoW Sensors” on page 5-1055. The Simulation 3D Rotorcraft block determines the vertical distance by calculating the starting and ending locations of the landing gear ray, depending on whether the aircraft is wheeled or nonwheeled.

Wheeled Aircraft

- Starting location

  The block calculates the starting location of the landing gear ray using the equation:

  starting_location = *representative_bone* - *Z* offset specified in block

Representative Bone	Z Offset Parameter
`NOSE_WHEEL`	**Front tire radius**
`LEFT_WHEEL`	**Left tire radius**
`RIGHT_WHEEL`	**Right tire radius**

- Ending location

  The block calculates the ending location using the equation:

  ending_location = *starting_location* - **Length of rays** or the point of first contact with a surface in the Unreal scene.

Nonwheeled Aircraft

- Starting location of ground contacts

  [*X*,*Y*,*Z*] offsets from the rotorcraft `BODY` bone location, in the NED frame. Specify these offsets with the parameters:

  - **Ground contact location 1**
  - **Ground contact location 2**
  - **Ground contact location 3**
  - **Ground contact location 4**
- Ending location

  The block calculates the ending location using the equation:

  ending_location = *starting_location* - **Length of rays** or the point of first contact with a surface in the Unreal scene.

How Altitude Sensors Work

The block determines the altitude with this workflow:

1. The block gives the ray starting point, direction, and maximum length to the altitude sensor, which exists in an Unreal Engine scene.
2. The Unreal Engine projects the ray from the starting point in the specified direction and determines if an object is within **Length of rays**.
3. Unreal Engine returns to the block information such as:
   - If an object is detected or not
   - If an object is detected, the location of the object
4. Using this information, the block calculate the altitude sensor values.

WoW Sensors

Aircraft use WoW switches to enable and disable systems. Weight-on-Wheels (WoW) sensors return a logical value of `1` (true) if one or more of the landing gear wheels touches a surface, `0` (false) otherwise. For example, allow landing gear retraction only when the WoW sensor returns `0` (false).

Wheeled Aircraft

To determine if any main landing gear tires touches the ground or any object, the WoW sensor uses the third and fourth rays of the altitude sensor. The left and right tire radius offsets must equal the actual radii of the tire meshes.

- If the sensed ground location is within ±**WoW sensor tolerance** of a ground contact point, the block returns a WoW sensor output of `1` (logical true).
- Otherwise, the block returns `0` (logical false).

Nonwheeled Aircraft

To determine if any one of the four ground contact points (**Ground contact location 1**, **Ground contact location 2**, **Ground contact location 3**, **Ground contact location 4**) are touching the ground, the WoW sensor uses the four ground contact location rays.

- If the sensed ground location is within ±**WoW sensor tolerance** of the tire surface, the block returns a WoW sensor output of `1` (logical true).
- Otherwise, the block returns `0` (logical false).

Version History

Introduced in R2023a

R2024a: Requires Simulink 3D Animation

Simulating models with the Simulation 3D Rotorcraft block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Rotorcraft to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

R2024a: Simulation 3D Rotorcraft Block Supports Single-Precision Inputs

Behavior changed in R2024a

The Simulation 3D Rotorcraft block now supports single- and double-precision inputs for the **Translation** and **Rotation** input ports. Prior to R2024a, this block supported only double-precision inputs for these ports.

See Also

Blocks
Simulation 3D Aircraft | Simulation 3D Actor Transform Get | Simulation 3D Actor Transform Set | Simulation 3D Camera Get | Simulation 3D Scene Configuration | Simulation 3D Message Get | Simulation 3D Message Set

Tools
Helicopter | Light Helicopter | Multirotor

Simulation 3D Sky Hogg Pack

Generate translation and rotation information for Sky Hogg

Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D Sky Hogg Pack block creates translation and rotation information for the Simulation 3D Aircraft block with **Type** set to `Sky Hogg`. Use the Simulation 3D Sky Hogg Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Aircraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Propeller_R — Propeller rotation
1-by-3 matrix

Propeller rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Propeller** parameter.

Data Types: `single` | `double`

Rudder_R — Rudder rotation
1-by-3 matrix

Rudder rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Rudder rotation** parameter.

Data Types: `single` | `double`

Elevator_R — Elevator rotation
1-by-3 matrix

Elevator rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Elevator rotation** parameter.

Data Types: `single` | `double`

Left_Aileron_R — Left aileron rotation
1-by-3 matrix

Left aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left aileron rotation** parameter.

Data Types: `single` | `double`

Right_Aileron_R — Right aileron rotation
1-by-3 matrix

Right aileron rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right aileron rotation** parameter.

Data Types: `single` | `double`

Flaps_R — Flaps rotation
1-by-3 matrix

Flaps rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Flaps rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_Strut_R — Nose wheel strut rotation
1-by-3 matrix

Nose wheel strut rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel strut rotation** parameter.

Data Types: `single` | `double`

Nose_Wheel_R — Nose wheel rotation
1-by-3 matrix

Nose wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Nose wheel rotation** parameter.

Data Types: `single` | `double`

Left_Wheel_R — Left wheel rotation
1-by-3 matrix

Left wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Left wheel rotation** parameter.

Data Types: `single` | `double`

Right_Wheel_R — Right wheel rotation
1-by-3 matrix

Right wheel rotation, specified as a 1-by-3 matrix.

Dependencies

To enable this port, select the **Right wheel rotation** parameter.

Data Types: `single` | `double`

Output

Translation — Aircraft translation
`11`-by-`3` array

Aircraft translation for Sky Hogg, returned as an `11`-by-`3` array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the aircraft.

The translation applies to these bones of the `Sky Hogg` type.

Bone	Index
BODY	1
PROPELLER	2
RUDDER	3
ELEVATOR	4
LEFT_AILERON	5
RIGHT_AILERON	6
FLAPS	7
NOSE_WHEEL_STRUT	8
NOSE_WHEEL	9
LEFT_WHEEL	10
RIGHT_WHEEL	11

Rotation — Aircraft rotation
11-by-3 array

Aircraft rotation for Sky Hogg aircraft, returned as a 11-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the aircraft.

Parameters

Propulsion

Propeller rotation — Option to enable Propeller_R input port

`off` (default) | `on`

Select this parameter to enable the **Propeller_R** input port.

Programmatic Use
Block Parameter: `Propeller_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Controls

Rudder rotation — Option to enable Rudder_R input port

`off` (default) | `on`

Select this parameter to enable the **Rudder_R** input port.

Programmatic Use
Block Parameter: `Rudder_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Elevator rotation — Option to enable Elevator_R input port

`off` (default) | `on`

Select this parameter to enable the **Elevator_R** input port.

Programmatic Use
Block Parameter: `Elevator_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Left aileron rotation — Option to enable Left_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Aileron_R** input port.

Programmatic Use
Block Parameter: `Left_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right aileron rotation — Option to enable Right_Aileron_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Aileron_R** input port.

Programmatic Use
Block Parameter: `Right_Aileron_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Flaps rotation — Option to enable Flaps_R input port

`off` (default) | `on`

Select this parameter to enable the **Flaps_R** input port.

Programmatic Use
Block Parameter: `Flaps_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Gear

Nose wheel strut rotation — Option to enable Nose_Wheel_Strut_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_Strut_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_Strut_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Nose wheel rotation — Option to enable Nose_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Nose_Wheel_R** input port.

Programmatic Use
Block Parameter: `Nose_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`

Default: `'off'`

Left wheel rotation — Option to enable Left_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Left_Wheel_R** input port.

Programmatic Use
Block Parameter: `Left_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Right wheel rotation — Option to enable Right_Wheel_R input port

`off` (default) | `on`

Select this parameter to enable the **Right_Wheel_R** input port.

Programmatic Use
Block Parameter: `Right_Wheel_R`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Dependencies

To enable this parameter, set **Select input coordinate frame** to `NED`.

Programmatic Use
Block Parameter: `AdjustForCesium`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use
Block Parameter: `Heading0`
Type: character vector
Values: `90` | real scalar
Default: `'90'`

Version History

Introduced in R2023b

R2024a: Simulation 3D Sky Hogg Pack Block Now Supports Earth-centered Earth-fixed (ECEF) Input Coordinate Frame

Behavior changed in R2024a

To enable ECEF coordinate systems for translation input, the Simulation 3D Sky Hogg Pack block has been updated. Use these parameters with the **Use Earth center as origin (ECEF)** parameter in the Simulation 3D Scene Configuration block.

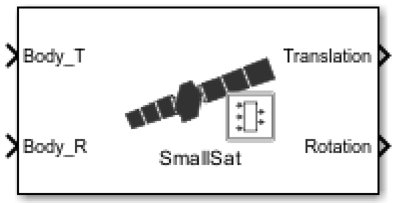
- **Select input coordinate frame**
  - **ECEF** — ECEF input coordinate frame. When you select this coordinate frame, the annotation `Origin: Earth center (ECEF)` appears under the pack block icons.
  - **NED** — NED coordinate frame. Prior to R2024a, NED was the only supported coordinate frame.
- **Direction of x-axis (degrees clockwise from north)** — Direction of the *x*-axis. This parameter is enabled when you select the **Enable geospatial correction** check box.

See Also

Simulation 3D Aircraft | Simulation 3D Scene Configuration

Simulation 3D SmallSat Pack

Create translation and rotation input matrices for SmallSat



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

The Simulation 3D SmallSat Pack block creates translation and rotation information for the Simulation 3D Spacecraft block with **Type** set to `SmallSat`. Use the Simulation 3D SmallSat Pack block to provide translation and rotation information to the **Translation** and **Rotation** input ports of the Simulation 3D Spacecraft block.

Ports

Input

Body_T — Body translation
1-by-3 matrix

Body translation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Body_R — Body rotation
1-by-3 matrix

Body rotation, specified as a 1-by-3 matrix.

Data Types: `single` | `double`

Output

Translation — Spacecraft translation
20-by-3 array

Spacecraft translation for SmallSat, returned as a 20-by-3 array. The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the spacecraft.

The translation applies to these bones of the `SmallSat` type.

Bone	Index
BODY	1
ANTENNA1	2

Bone	Index
ANTENNA2	3
ANTENNA3	4
ANTENNA4	5
SENSOR1	6
SENSOR2	7
SENSOR3	8
SENSOR4	9
SOLAR_ARRAY1	10
SOLAR_ARRAY1_1	11
SOLAR_ARRAY1_1_1	12
SOLAR_ARRAY1_1_1_1	13
SOLAR_ARRAY1_1_1_1_1	14
SOLAR_ARRAY2	15
SOLAR_ARRAY2_2	16
SOLAR_ARRAY2_2_2	17
SOLAR_ARRAY2_2_2_2	18
SOLAR_ARRAY2_2_2_2_2	19
THRUSTER	20

Rotation — Spacecraft rotation
20-by-3 array

Spacecraft rotation for SmallSat, returned as a 20-by-3 array.

The rotation applies to the same bones as listed for the “Translation” on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the spacecraft.

Parameters

To edit block parameters interactively, use the Property Inspector. From the Simulink Toolstrip, on the **Simulation** tab, in the **Prepare** gallery, select **Property Inspector**.

Sensors

Antenna 1 translation — Option to enable Antenna1_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna1_T`
Values:	`'off'` (default) | `'on'`

Antenna 1 rotation — Option to enable Antenna1_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna1_R`
Values:	`'off'` (default) | `'on'`

Antenna 2 translation — Option to enable Antenna2_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna2_T`
Values:	`'off'` (default) | `'on'`

Antenna 2 rotation — Option to enable Antenna2_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna2_R`
Values:	`'off'` (default) | `'on'`

Antenna 3 translation — Option to enable Antenna3_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna3_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna3_T`
Values:	`'off'` (default) | `'on'`

Antenna 3 rotation — Option to enable Antenna3_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna3_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna3_R`
Values:	`'off'` (default) | `'on'`

Antenna 4 translation — Option to enable Antenna4_T input port

`off` (default) | `on`

Select this parameter to enable the **Antenna4_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna4_T`
Values:	`'off'` (default) | `'on'`

Antenna 4 rotation — Option to enable Antenna4_R input port

`off` (default) | `on`

Select this parameter to enable the **Antenna4_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Antenna4_R`
Values:	`'off'` (default) | `'on'`

Sensor 1 translation — Option to enable Sensor1_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor1_T`
Values:	`'off'` (default) | `'on'`

Sensor 1 rotation — Option to enable Sensor1_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor1_R`
Values:	`'off'` (default) | `'on'`

Sensor 2 translation — Option to enable Sensor2_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor2_T`
Values:	`'off'` (default) | `'on'`

Sensor 2 rotation — Option to enable Sensor2_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor2_R`

Values:	`'off'` (default) | `'on'`

Sensor 3 translation — Option to enable Sensor3_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor3_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor3_T`
Values:	`'off'` (default) | `'on'`

Sensor 3 rotation — Option to enable Sensor3_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor3_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor3_R`
Values:	`'off'` (default) | `'on'`

Sensor 4 translation — Option to enable Sensor4_T input port

`off` (default) | `on`

Select this parameter to enable the **Sensor4_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor4_T`
Values:	`'off'` (default) | `'on'`

Sensor 4 rotation — Option to enable Sensor4_R input port

`off` (default) | `on`

Select this parameter to enable the **Sensor4_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Sensor4_R`
Values:	`'off'` (default) | `'on'`

Solar Arrays

Solar array 1 translation — Option to enable SolarArray1_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1 rotation — Option to enable SolarArray1_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_R`
Values:	`'off'` (default) | `'on'`

Solar array 1.1 translation — Option to enable SolarArray1_1_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1.1 rotation — Option to enable SolarArray1_1_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_R`
Values:	`'off'` (default) | `'on'`

Solar array 1.1.1 translation — Option to enable SolarArray1_1_1_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1.1.1 rotation — Option to enable SolarArray1_1_1_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_1_R`
Values:	`'off'` (default) | `'on'`

Solar array 1.1.1.1 translation — Option to enable SolarArray1_1_1_1_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray1_1_1_1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_1_1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1.1.1.1 rotation — Option to enable SolarArray1_1_1_1_R input port

off (default) | on

Select this parameter to enable the **SolarArray1_1_1_1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_1_1_R`
Values:	`'off'` (default) | `'on'`

Solar array 1.1.1.1.1 translation — Option to enable SolarArray1_1_1_1_1_T input port

off (default) | on

Select this parameter to enable the **SolarArray1_1_1_1_1_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_1_1_1_T`
Values:	`'off'` (default) | `'on'`

Solar array 1.1.1.1.1 rotation — Option to enable SolarArray1_1_1_1_1_R input port

off (default) | on

Select this parameter to enable the **SolarArray1_1_1_1_1_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray1_1_1_1_1_R`
Values:	`'off'` (default) | `'on'`

Solar array 2 translation — Option to enable SolarArray2_T input port

off (default) | on

Select this parameter to enable the **SolarArray2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_T`

Values:	`'off'` (default) | `'on'`

Solar array 2 rotation — Option to enable SolarArray2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_R`
Values:	`'off'` (default) | `'on'`

Solar array 2.2 translation — Option to enable SolarArray2_2_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_T`
Values:	`'off'` (default) | `'on'`

Solar array 2.2 rotation — Option to enable SolarArray2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_R`
Values:	`'off'` (default) | `'on'`

Solar array 2.2.2 translation — Option to enable SolarArray2_2_2_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_2_T`
Values:	`'off'` (default) | `'on'`

Solar array 2.2.2 rotation — Option to enable SolarArray2_2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_2_R`
Values:	`'off'` (default) | `'on'`

Solar array 2.2.2.2 translation — Option to enable SolarArray2_2_2_2_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_2_2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_2_2_T`
Values:	`'off'` (default) | `'on'`

Solar array 2.2.2.2 rotation — Option to enable SolarArray2_2_2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_2_2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_2_2_R`
Values:	`'off'` (default) | `'on'`

Solar array 2.2.2.2.2 translation — Option to enable SolarArray2_2_2_2_2_T input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_2_2_2_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_2_2_2_T`
Values:	`'off'` (default) | `'on'`

Solar array 2.2.2.2.2 rotation — Option to enable SolarArray2_2_2_2_2_R input port

`off` (default) | `on`

Select this parameter to enable the **SolarArray2_2_2_2_2_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SolarArray2_2_2_2_2_R`
Values:	`'off'` (default) | `'on'`

Thrusters

Thruster translation — Option to enable Thruster_T input port

`off` (default) | `on`

Select this parameter to enable the **Thruster_T** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Thruster_T`
Values:	`'off'` (default) | `'on'`

Thruster rotation — Option to enable Thruster_R input port

`off` (default) | `on`

Select this parameter to enable the **Thruster_R** input port.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Thruster_R`
Values:	`'off'` (default) | `'on'`

Geospatial

Select input coordinate frame — Input coordinate frame

`NED` (default) | `ECEF`

Select the input coordinate frame to position aerospace vehicles with respect to north-east-down (NED) or the Earth center (ECEF). Use this parameter with the **Use Earth center as origin (ECEF)** parameter of the Simulation 3D Scene Configuration block.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`GeoRadioButton`
Values:	`'NED'` (default) | `'ECEF'`

Enable geospatial correction — Option to enable geospatial correction

`off` (default) | `on`

Select this parameter to enable the geospatial correction to the input body translation and rotation.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`AdjustForCesium`
Values:	`'off'` (default) | `'on'`

Direction of x-axis (degrees clockwise from north) — Compass direction of *x*-axis

`90` (default) | real scalar

Specify the compass direction of the *x*-axis, specified as a real scalar, in degrees.

Dependencies

To enable this parameter:

- Set **Select input coordinate frame** to **NED**
- Select **Enable geospatial correction**

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Heading0`
Values:	`'90'` (default) | real scalar

Version History

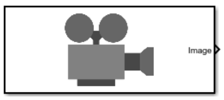
Introduced in R2024a

See Also

Simulation 3D Spacecraft | SmallSat | CubeSat Vehicle

Simulation 3D Camera Get

Camera image



Libraries:
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core
Aerospace Blockset / Animation / Simulation 3D
Simulink 3D Animation / Simulation 3D / Utilities

Description

Note Simulating models with the Simulation 3D Camera Get block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Camera Get to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Camera Get block provides an interface to an ideal camera in the 3D visualization environment. The image output is a red, green, and blue (RGB) array.

If you set the sample time to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, ensure that the Simulation 3D Scene Configuration block is in your model.

Tip Verify that the Simulation 3D Scene Configuration block executes before the Simulation 3D Camera Get block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Camera Get block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Camera Get — `1`

For more information about execution order, see “Control and Display Execution Order”.

Ports

Output

Image — 3D output camera image
m-by-*n*-by-3 array of RGB triplet values

3D output camera image, returned as an *m*-by-*n*-by-3 array of RGB triplet values. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

Data Types: `int8` | `uint8`

Parameters

Mounting

Sensor identification — Number to identify unique sensor

`0` (default) | positive integer

Unique sensor identifier, specified as a positive integer. This number is used to identify a specific sensor. The sensor identifier distinguishes between sensors in a multi-sensor system.

Example: 2

Vehicle name — Name of a vehicle

`Scene Origin` (default) | character vector

Vehicle name. Block provides a list of vehicles in the model. If you select `Scene Origin`, the block places a sensor at the scene origin.

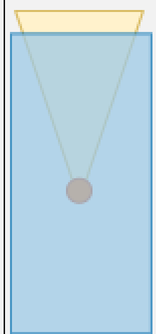
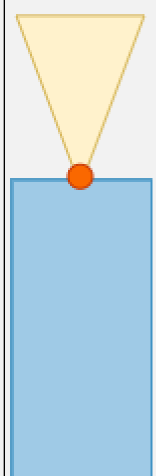
Example: `SimulinkVehicle1`

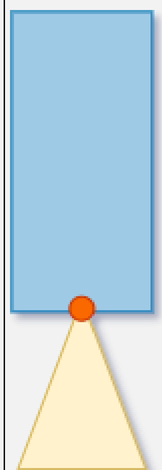
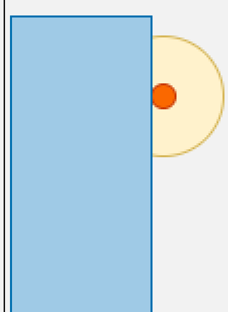
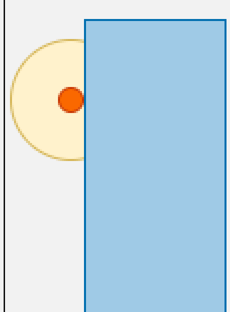
Vehicle mounting location — Sensor mounting location

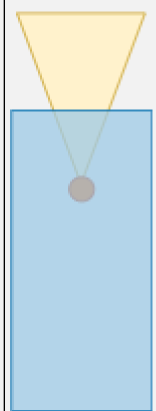
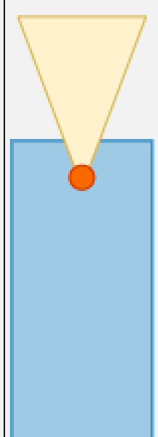
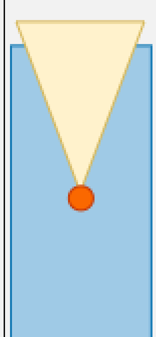
`Origin` (default) | `Front bumper` | `Rear bumper` | `Right mirror` | `Left mirror` | `Rearview mirror` | `Hood center` | `Roof center`

Sensor mounting location.

- When **Vehicle name** is `Scene Origin`, the block mounts the sensor to the origin of the scene, and **Mounting location** can be set to `Origin` only. During simulation, the sensor remains stationary.
- When **Vehicle name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground and at the geometric center of the vehicle 	[0, 0, 0]
`Front bumper`	Forward-facing sensor mounted to the front bumper 	[0, 0, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rear bumper`	Backward-facing sensor mounted to the rear bumper 	[0, 0, 180]
`Right mirror`	Downward-facing sensor mounted to the right side-view mirror 	[0, -90, 0]
`Left mirror`	Downward-facing sensor mounted to the left side-view mirror 	[0, -90, 0]

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Rearview mirror`	Forward-facing sensor mounted to the rearview mirror, inside the vehicle 	[0, 0, 0]
`Hood center`	Forward-facing sensor mounted to the center of the hood 	[0, 0, 0]
`Roof center`	Forward-facing sensor mounted to the center of the roof 	[0, 0, 0]

The (*X*, *Y*, *Z*) location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D Scene Configuration block to which you are mounting. The tables show the *X*, *Y*, and *Z* locations of sensors in the vehicle coordinate system. In this coordinate system:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up from the ground.
- Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, then the yaw angle (that is, the orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

Box Truck — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	5.10	0	0.60
`Rear bumper`	–5	0	0.60
`Right mirror`	2.90	1.60	2.10
`Left mirror`	2.90	–1.60	2.10
`Rearview mirror`	2.60	0.20	2.60
`Hood center`	3.80	0	2.10
`Roof center`	1.30	0	4.20

Hatchback — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	1.93	0	0.51
`Rear bumper`	–1.93	0	0.51
`Right mirror`	0.43	–0.84	1.01
`Left mirror`	0.43	0.84	1.01
`Rearview mirror`	0.32	0	1.27
`Hood center`	1.44	0	1.01
`Roof center`	0	0	1.57

Muscle Car — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	2.47	0	0.45
`Rear bumper`	–2.47	0	0.45
`Right mirror`	0.43	–1.08	1.01
`Left mirror`	0.43	1.08	1.01
`Rearview mirror`	0.32	0	1.20
`Hood center`	1.28	0	1.14
`Roof center`	–0.25	0	1.58

Sedan — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	2.42	0	0.51
`Rear bumper`	–2.42	0	0.51
`Right mirror`	0.59	–0.94	1.09
`Left mirror`	0.59	0.94	1.09
`Rearview mirror`	0.43	0	1.31
`Hood center`	1.46	0	1.11
`Roof center`	–0.45	0	1.69

Small Pickup Truck — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	3.07	0	0.51
`Rear bumper`	–3.07	0	0.51
`Right mirror`	1.10	–1.13	1.52
`Left mirror`	1.10	1.13	1.52
`Rearview mirror`	0.85	0	1.77
`Hood center`	2.22	0	1.59
`Roof center`	0	0	2.27

Sport Utility Vehicle — Sensor Locations Relative to Vehicle Origin

Mounting Location	X (m)	Y (m)	Z (m)
`Front bumper`	2.42	0	0.51
`Rear bumper`	–2.42	0	0.51
`Right mirror`	0.60	–1	1.35
`Left mirror`	0.60	1	1.35
`Rearview mirror`	0.39	0	1.55
`Hood center`	1.58	0	1.39
`Roof center`	–0.56	0	2

Example: `Origin`

Specify offset — Specify offset from mounting location

`off` (default) | `on`

Select this parameter to specify an offset from the mounting location.

Relative translation [X, Y, Z] — Translation offset from mounting location

`[0,0,0]` (default) | real-valued 1-by-3 vector

Specify a translation offset from the mount location, about the vehicle coordinate system *X*, *Y*, and *Z* axes. Units are in meters.

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when facing forward.
- The *Z*-axis points up.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

Relative rotation [Roll, Pitch, Yaw] — Rotational offset from mounting location

`[0,0,0]` (default) | real-valued 1-by-3 vector

Specify a rotational offset from the mounting location, about the vehicle coordinate system *X*, *Y*, and *Z* axes. Units are in degrees.

- Roll angle is the angle of rotation about the *X*-axis of the vehicle coordinate system. A positive roll angle corresponds to a clockwise rotation when looking in the positive direction of the *X*-axis.
- Pitch angle is the angle of rotation about the *Y*-axis of the vehicle coordinate system. A positive pitch angle corresponds to a clockwise rotation when looking in the positive direction of the *Y*-axis.
- Yaw angle is the angle of rotation about the *Z* of the vehicle coordinate system. A positive yaw angle corresponds to a clockwise rotation when looking in the positive direction of the *Z*-axis.

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

Sample time — Sample time

`-1` (default) | positive scalar

Sample time of the block in seconds. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

Parameter

Horizontal resolution — Pixels

`uint32(1280)` (default) | `scalar`

Horizontal image resolution, in pixels.

Vertical resolution — Pixels

`uint32(720)` (default) | `scalar`

Vertical image resolution, in pixels.

Horizontal field of view — Field of view

`single(60)` (default) | `scalar`

Horizontal field of view (FOV), in deg.

Tips

- To understand how to set tag of **Sim 3d Scene Cap** and how it the tag is related to the Simulation 3D Camera Get block, see "Place Cameras on Actors in the Unreal Editor" (Vehicle Dynamics Blockset).

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

Behavior change in future release

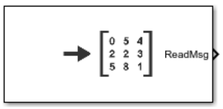
Simulating models with the Simulation 3D Camera Get block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Camera Get to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

See Also

Simulation 3D Scene Configuration

Simulation 3D Message Get

Retrieve data from Unreal Engine visualization environment



Libraries:
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core
Aerospace Blockset / Animation / Simulation 3D

Description

Note Simulating models with the Simulation 3D Message Get block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Message Get to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Message Get block retrieves data from the Unreal Engine 3D visualization environment. In your model, ensure that the Simulation 3D Scene Configuration block is at the same level as the Simulation 3D Message Get block.

Tip Verify that the Simulation 3D Scene Configuration block executes before the Simulation 3D Message Get block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Message Get block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Message Get — `1`

For more information about execution order, see “Control and Display Execution Order”.

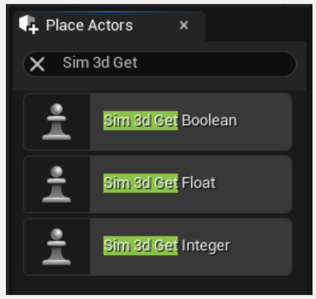
Configure Scenes to Send Data

To use the block, you must configure scenes in the Unreal Engine environment to send data to the Simulink model:

1. Install the customize 3D scenes for aerospace simulations.
2. In the Unreal Editor, follow these general workflows to send data to Simulink.

<table>
<tr><th>Unreal Engine User</th><th>Workflow</th></tr>
<tr><td>Blueprint</td><td>

a Instantiate the `Sim3DSet` actor that corresponds to the data type you want to send to the Simulink model. This example shows the Unreal Editor `Sim3DSet` data types.

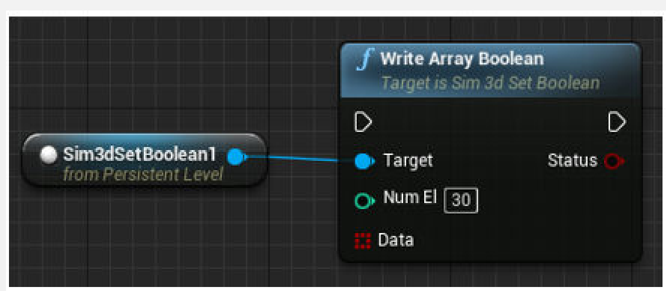


b Specify an actor tag name that matches the Simulation 3D Message Get block **Signal name** parameter.

c Navigate to the Level Blueprint.

d Find the blueprint method for the `Sim3DSet` actor class based on the data type and size specified by the Simulation 3D Message Get block **Data type** and **Message size** parameters.

For example, in Unreal Editor, this diagram shows that `Write Array Boolean` is the method for the `Sim3DSetBoolean` actor class that sends Boolean data type of array size 30.



e Compile and save the scene.

Note By default, the Double Lane Change scene has a `Sim3DSetBoolean` actor with tag name `NumOfConesHit`.

</td></tr>
</table>

Unreal Engine User	Workflow
C++ class	**a** Create a new actor class for the mesh or asset that you want the Simulink model to interact with. Derive it from `ASim3dActor`. **b** In the new actor class: • Declare a pointer to the signal name as a class field. • Get the class tag. • Create a signal writer and assign the pointer in the method `Sim3dSetup`. • In the method `Sim3dStep`, invoke the `WriteSimulation3DMessage` function to write the data to the Simulink model. • Delete the signal writer in the method `Sim3dRelease` of the actor.

For more information about the Unreal Editor, see the Unreal Engine Documentation.

Ports

Output

ReadMsg — Data retrieved from scene
`scalar` | `array`

Data retrieved from the 3D visualization environment scene data. In the Unreal Engine environment, you can use the `Sim3DSet` class to configure scene actors to send data to the Simulink model.

Parameters

Signal name, SigName — Message signal name

`mySignal` (default)

Specifies the signal name in the 3D visualization environment. In the Unreal Engine environment, use the `Sim3DSet` actor class 'Tags' property located in the 'Details' pane.

For example, you can retrieve data from the double-lane change scene that indicates if cones are hit during a double-lane change maneuver. To retrieve cone hit data from the double-lane change scene, set this parameter to `NumOfConesHit`. In the double-lane change scene, the `Sim3DSet` actor class 'Tags' property is set to `NumOfConesHit`.

Data type, DataType — Message data type

`double*` | `single` | `int8*` | `uint8*` | `int16*` | `uint16*` | `int32` | `uint32*` | `boolean`

3D visualization environment signal data type. The supported data types depend on the Unreal Engine workflow.

Workflow	Supported Data Types
Blueprint	single int32 Boolean
*C++ class	double single int8 uint8 int16 uint16 int32 uint32 Boolean

In the Unreal Engine environment, instantiate the Sim3DSet actor class for the data type that you want to send to the Simulink model. For example, you can retrieve data from the double-lane change scene that indicates if cones are hit during a double-lane change maneuver. To retrieve cone hit data from the double-lane change scene, set this parameter to boolean. In the double-lane change scene, the Sim3DSetBoolean actor class is instantiated to send the cone hit or miss boolean data.

Message size, MsgSize — Message dimension

[1 1] (default) | scalar | array

3D visualization environment signal dimension. In the Unreal Engine environment blueprint, set the input to the node of the Sim3DSet actor class to specify the dimensions of data that you want to send to the Simulink model.

For example, you can retrieve data from the double-lane change scene that indicates if cones are hit during a double-lane change maneuver. To retrieve cone hit data from the double-lane change scene, set this parameter to [2 15]. In the double-lane change scene, the input to the blueprint node for the Sim3DSetBoolean actor class is set to 30, the number of cones in the scene.

Sample time — Sample time

0.02 (default) | -1 | scalar

Sample time, in s. The graphics frame rate is the inverse of the sample time. If you set the sample time to -1, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation
Behavior change in future release

Simulating models with the Simulation 3D Message Get block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Message Get to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

See Also

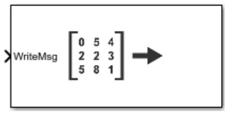
Simulation 3D Scene Configuration | Simulation 3D Message Set

External Websites

Unreal Engine

Simulation 3D Message Set

(To be removed) Send data to Unreal Engine visualization environment



Libraries:
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core
Aerospace Blockset / Animation / Simulation 3D
Simulink 3D Animation / Simulation 3D / Utilities

Description

Note Simulating models with the Simulation 3D Message Set block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Message Set to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Message Set block sends data to the Unreal Engine 3D visualization environment. In your model, ensure that the Simulation 3D Scene Configuration block is at the same level as the Simulation 3D Message Set block.

Tip Verify that the Simulation 3D Message Set block executes before the Simulation 3D Scene Configuration block. That way, Simulation 3D Message Set prepares the signal data before the Unreal Engine 3D visualization environment receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Message Set — `-1`

For more information about execution order, see “Control and Display Execution Order”.

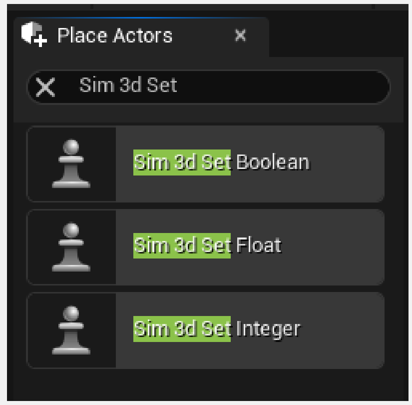
Configure Scenes to Receive Data

To use the block, you must configure scenes in the Unreal Engine environment to receive data from the Simulink model:

1. Install the customize 3D scenes for aerospace simulations.
2. In the Unreal Editor, follow these general workflows to receive data from Simulink.

<table>
<tr><th>Unreal Engine User</th><th>Workflow</th></tr>
<tr><td>Blueprint</td><td>

a Instantiate the `Sim3DGet` actor that corresponds to the data type you want to receive from the Simulink model. This example shows the Unreal Editor `Sim3DGet` data types.

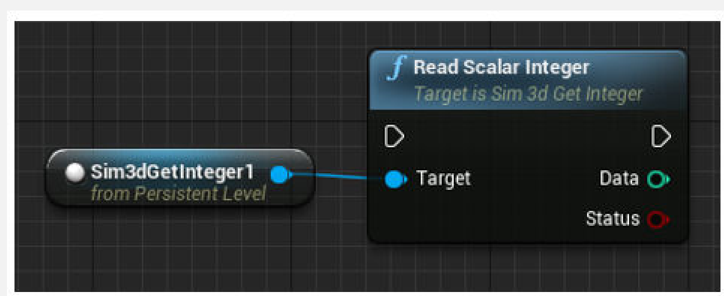


b Specify an actor tag name that matches the Simulation 3D Message Set block **Signal name** parameter.

c Navigate to the Level Blueprint.

d Find the blueprint method for the `Sim3DGet` actor class based on the data type and size that you want to receive from the Simulink model.

For example, in Unreal Editor, this diagram shows that `Read Scalar Integer` is the method for `Sim3DGetInteger` actor class to receive `int32` data type of size scalar.



e Compile and save the scene.

Note By default, the Double Lane Change scene has a `Sim3DGetInteger` actor with tag name `TrafficLight1`.

</td></tr>
</table>

Unreal Engine User	Workflow
C++ class	**a** Create a new actor class for the mesh or asset that you want the Simulink model to interact with. Derive it from `ASim3dActor`. **b** In the new actor class: • Declare a pointer to the signal name as a class field. • Get the class tag. • Create a signal reader and assign the pointer in the method `Sim3dSetup`. • In the method `Sim3dStep`, invoke the `ReadSimulation3DMessage` function to read the data from a Simulink model. • Delete the signal reader in the method `Sim3dRelease` of the actor.

For more information about the Unreal Editor, see the Unreal Engine Documentation.

Ports

Input

WriteMsg — Data sent to scene
`scalar` | `array`

Data sent to the 3D visualization environment scene. In the Unreal Engine environment, you can configure the `Sim3DGet` class to receive the data from the Simulink model.

Parameters

Signal name, SigName — Message signal name

`mySignal` (default)

Specifies the signal name in the 3D visualization environment. In the Unreal Engine environment, use the `Sim3Get` actor class 'Tags' property located in the 'Details' pane.

For example, you can send data to the double lane change scene that changes the traffic signal light color to red, yellow, or green. To send data to the traffic signal light, set this parameter to `TrafficLight1`. In the double lane change scene, the 'Tags' property value for `Sim3dGetInteger` actor class is set to TrafficLight1.

Sample time — Sample time

`0.02` (default) | `-1` | `scalar`

Sample time, in s. The graphics frame rate is the inverse of the sample time. If you set the sample time to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

Behavior change in future release

Simulating models with the Simulation 3D Message Set block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Message Set to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

R2024a: To be removed

Not recommended starting in R2024a

The Simulation 3D Message Set block will be removed in a future release. Instead, use Simulation 3D Actor.

See Also

Simulation 3D Scene Configuration | Simulation 3D Message Get

External Websites

Unreal Engine

Simulation 3D Scene Configuration

Scene configuration for 3D simulation environment

Libraries:
Simulink 3D Animation / Simulation 3D / Environment
Aerospace Blockset / Animation / Simulation 3D
Automated Driving Toolbox / Simulation 3D
Robotics System Toolbox / Simulation 3D
UAV Toolbox / Simulation 3D
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core

Description

Note Simulating models with the Simulation 3D Scene Configuration block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Scene Configuration to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Scene Configuration block implements a 3D simulation environment that is rendered by using the Unreal Engine from Epic Games. Aerospace Blockset Interface for Unreal Engine Projects integrates the 3D simulation environment with Simulink so that you can query the world around the vehicle and virtually test perception, control, and planning algorithms. Using this block, you can also control the position of the sun and the weather conditions of a scene. For more details, see Sun Position and Weather on page 5-1118.

You can simulate from a set of prebuilt scenes or from your own custom scenes. Scene customization requires the Aerospace Blockset Interface for Unreal Engine Projects support package. For more details, see “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2.

You can also simulate custom scenes designed and built in RoadRunner. To do so, you must first export scenes from RoadRunner and specify the path to the exported scene artifacts in the **Project** parameter of the Simulation 3D Scene Configuration block.

Tip The Simulation 3D Scene Configuration block must execute after blocks that send data to the 3D environment and before blocks that receive data from the 3D environment. To verify the execution order of such blocks, right-click the blocks and select **Properties**. Then, on the **General** tab, confirm these **Priority** settings:

- For blocks that send data to the 3D environment, such as Simulation 3D Vehicle with Ground Following blocks, **Priority** must be set to `-1`. That way, these blocks prepare their data before the 3D environment receives it.
- For the Simulation 3D Scene Configuration block in your model, **Priority** must be set to `0`.
- For blocks that receive data from the 3D environment, such as blocks, **Priority** must be set to `1`. That way, the 3D environment can prepare the data before these blocks receive it.

For more information about execution order, see “Control and Display Execution Order”.

Parameters

Scene

Scene Selection

Scene source — Source of scene

`Default Scenes` (default) | `RoadRunner` | `Unreal Executable` | `Unreal Editor`

Source of the scene in which to simulate, specified as one of the options in the table.

Option	Description
`Default Scenes`	Simulate in one of the default, prebuilt scenes specified in the **Scene name** parameter.
`RoadRunner`	Simulate in a RoadRunner scene. To import the scene, specify the RoadRunner file in the **Project** parameter.
`Unreal Executable`	Simulate in a scene that is part of an Unreal Engine executable file. Specify the executable file in the **Project name** parameter. Specify the scene in the **Scene** parameter. Select this option to simulate in custom scenes that have been packaged into an executable for faster simulation.
`Unreal Editor`	Simulate in a scene that is part of an Unreal Engine project (`.uproject`) file and is open in the Unreal Editor. Specify the project file in the **Project** parameter. Select this option when developing custom scenes. By clicking **Open Unreal Editor**, you can co-simulate within Simulink and the Unreal Editor and modify your scenes based on the simulation results.

Scene name — Name of prebuilt 3D scene

`Airport` (default)

Name of the prebuilt 3D scene in which to simulate, specified as one of these options.

The Aerospace Blockset Interface for Unreal Engine Projects contains customizable versions of these scenes. For details about customizing scenes, see “Customize Scenes Using Simulink and Unreal Editor” on page 4-7.

Dependencies

To enable this parameter, set **Scene source** to `Default Scenes`.

Project name — Name of Unreal Engine executable file

VehicleSimulation.exe (default) | valid executable file name

Name of the Unreal Engine executable file, specified as a valid executable project file name. You can either browse for the file or specify the full path to the project file, using backslashes. To specify a scene from this file to simulate in, use the **Scene** parameter.

By default, **Project name** is set to `VehicleSimulation.exe`, which is on the MATLAB search path.

Note If you select a custom Unreal executable file built with a version of Unreal Engine that is not compatible with the current version of MATLAB, you receive one of these error messages:

- `"Incompatible version of 3D Simulation engine: Undefined"`: For Unreal executable built for a MATLAB release prior to R2023b
- `"Incompatible version of 3D Simulation engine: 23.2.0"`: For Unreal executable built for MATLAB R2023b or later

To resolve these errors, you must migrate the Unreal project and rebuild the executable with the latest support package installed. For more information, see .

Example: `C:\Local\WindowsNoEditor\AutoVrtlEnv.exe`

Dependencies

To enable this parameter, set **Scene source** to `Unreal Executable`.

Scene — Name of scene from executable file

/Game/Maps/HwStrght (default) | path to valid scene name

Name of a scene from the executable file specified by the **Project name** parameter, specified as a path to a valid scene name.

When you package scenes from an Unreal Engine project into an executable file, the Unreal Editor saves the scenes to internal folders within the executable file. Default scenes are located at the path `/Game/Maps`. Aerospace Blockset scenes are located at the path `/MathWorksAerospaceContent/Maps`. Therefore, you must prepend either `/Game/Maps or /MathWorksAerospaceContent/Maps` to the scene name. Specify this path using forward slashes. For the file name, do not specify the `.umap` extension. For example, if the scene from the executable in which you want to simulate is named `Airport.umap`, specify **Scene** as `/MathWorksAerospaceContent/Maps/Airport`.

By default, **Scene** is set to `/Game/Maps/HwStrght`, which is a scene from the default `VehicleSimulation.exe` executable file specified by the **Project name** parameter. This scene corresponds to the prebuilt **Straight Road** scene.

Example: `/MathWorksAerospaceContent/Maps/Airport`

Dependencies

To enable this parameter, set **Scene source** to `Unreal Executable`.

Project — Name of Unreal Engine project file or RoadRunner file

valid project file name

Name of the Unreal Engine project file or RoadRunner file, specified as a valid project file name. You can either browse for the file or specify the full path to the file, using backslashes. The file name must contain no spaces.

To simulate scenes from Unreal Engine project file in the Unreal Editor, click **Open Unreal Editor**. If you have an Unreal Editor session open already, then this button is disabled.

To run the simulation, in Simulink, click **Run**. Before you click **Play** in the Unreal Editor, wait until the Diagnostic Viewer window displays this confirmation message:

```
In the Simulation 3D Scene Configuration block, you set the scene source to 'Unreal Editor'.
In Unreal Editor, select 'Play' to view the scene.
```

This message confirms that Simulink has instantiated the scene actors, including the vehicles and cameras, in the Unreal Engine 3D environment. If you click **Play** before the Diagnostic Viewer window displays this confirmation message, Simulink might not instantiate the actors in the Unreal Editor.

To simulate a RoadRunner scene, browse for the folder containing the exported RoadRunner scene files, then specify the Filmbox (`.fbx`) file.

Dependencies

To enable this parameter, set **Scene source** to `Unreal Editor` or `RoadRunner`. The `RoadRunner` option requires a Simulink 3D Animation license.

Scene Parameters

Scene view — Placement of virtual camera that displays scene

`Custom` | `Scene Origin` | vehicle name

Configure the placement of the virtual camera that displays the scene during simulation.

- If your model contains no vehicle blocks, then during simulation, you view the scene in free camera mode from a camera positioned at the scene origin. To change the view to a custom viewpoint, set **Scene view** to `Custom`. Then, set the **Initial viewer translation** and **Initial viewer rotation** values to view the scene in free camera mode at the specified viewpoint.
- If your model contains at least one vehicle block, then by default, you view the scene from a camera attached to the first vehicle that was placed in your model. To change the view to a different vehicle, set **Scene view** to the name of that vehicle. The **Scene view** parameter list is populated with all the **Name** parameter values of the vehicle blocks contained in your model.

If you add a Simulation 3D Scene Configuration block to your model before adding any vehicle blocks, the virtual camera remains positioned at the scene origin. To reposition the camera to follow a vehicle, update this parameter.

Scene Controls

When **Scene view** is set to `Scene Origin` or `Custom`, you can change the location of the camera to navigate in the scene during simulation using keyboard and mouse control. You can also change the camera location when the simulation is paused.

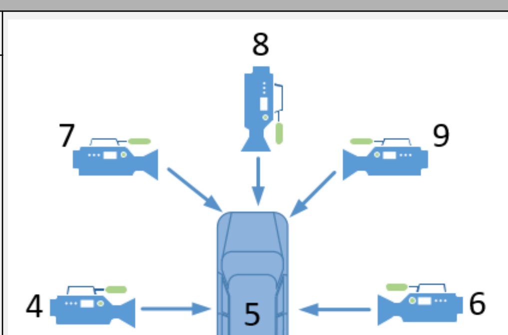
To navigate in 3D environment, use these keyboard shortcuts.

Keyboard Shortcut	Camera Control
W	Move forward.
Shift+W	Move faster in the forward direction.
S	Move backward.
Shift+S	Move faster in reverse direction.
A	Move left.
Shift+A	Move faster toward left.
D	Move right.
Shift+D	Move faster toward right.
Q	Move up.
Shift+Q	Move faster in the upward direction.
E	Move down.
Shift+E	Move faster in the downward direction.
Mouse scroll wheel	Control the camera distance from the cursor point.
Shift+mouse scroll wheel	Camera moves faster.
Mouse right-click and drag	Orbit around the cursor point.
O	Attach camera to actor under the cursor point.
Mouse left-click and drag	Orbit around the actor after camera attaches to the actor.
P	Detach camera from the actor.
L	Record the current viewpoint and display `Location saved` on the scene.
1 to 9	Access the stored viewpoints, if available.
Tab	Cycle the view between all vehicles in the scene and the viewpoint at the start of the simulation, in the forward direction.
Shift+Tab	Cycle the view between all vehicles in the scene and the viewpoint at the start of the simulation, in the reverse direction.
I	Display the keyboard shortcuts on the screen.

Vehicle Views

When **Scene view** is set to a vehicle name, during simulation, you can change the location of the camera around the vehicle. You can also set the view to a vehicle during simulation by pressing **Tab**.

To smoothly change the camera views, use these keyboard shortcuts.

Keyboard Shortcut	Camera View	
1	Back left	
2	Back	

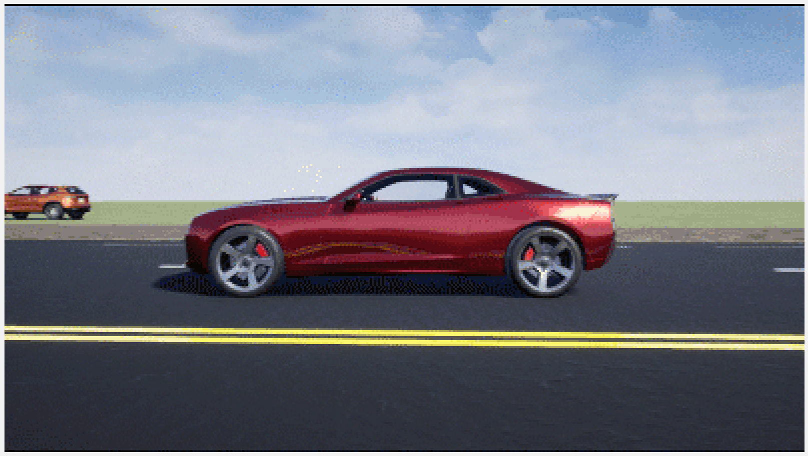
Keyboard Shortcut	Camera View	
3	Back right	**View Animated GIF** 
4	Left	
5	Internal	
6	Right	
7	Front left	
8	Front	
9	Front right	
0	Overhead	

For additional camera controls, use these keyboard shortcuts.

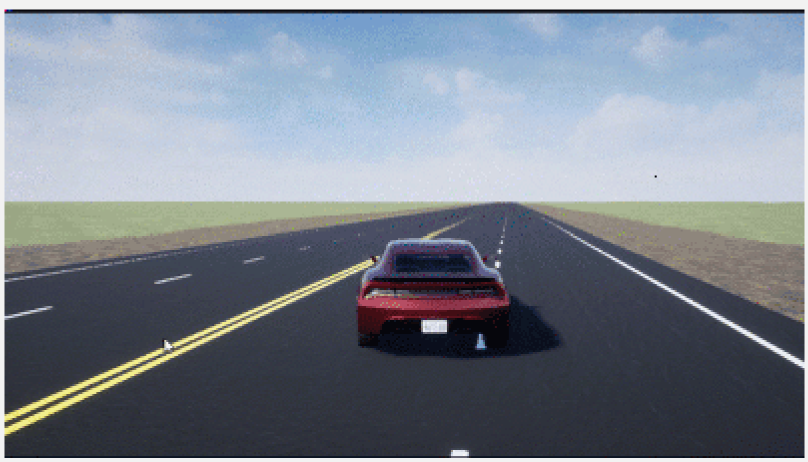
Keyboard Shortcut	Camera Control
Tab	Cycle the view between all vehicles in the scene. **View Animated GIF** 

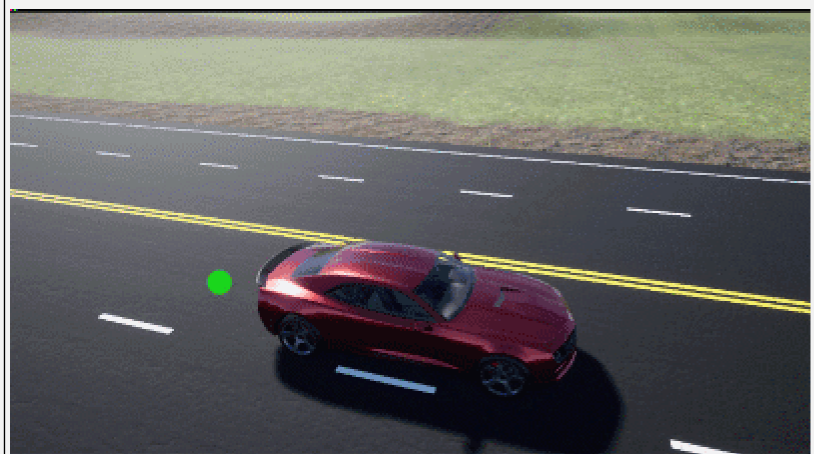
<table>
<tr><th>Keyboard Shortcut</th><th>Camera Control</th></tr>
<tr><td>Mouse scroll wheel</td><td>Control the camera distance from the vehicle.

View Animated GIF

</td></tr>
<tr><td>L</td><td>Toggle a camera lag effect on or off. When you enable the lag effect, the camera view includes:
Position lag, based on the vehicle translational acceleration
Rotation lag, based on the vehicle rotational velocity
This lag improves visualization of overall vehicle acceleration and rotation.

View Animated GIF

</td></tr>
</table>

Keyboard Shortcut	Camera Control
F	Toggle the free camera mode on or off. When you enable the free camera mode, you can use the mouse to change the pitch and yaw of the camera. This mode allows you to orbit the camera around the vehicle. **View Animated GIF** 

Sample time — Sample time of visualization engine

`.02` (default) | real positive scalar

Sample time, T_s, of the visualization engine, specified as a real positive scalar. Units are in seconds.

The graphics frame rate of the visualization engine is the inverse of the sample time. For example, if **Sample time** is `1/60`, then the visualization engine solver tries to achieve a frame rate of 60 frames per second. However, the real-time graphics frame rate is often lower due to factors such as graphics card performance and model complexity.

By default, blocks that receive data from the visualization engine inherit this sample rate.

Display 3D simulation window — Unreal Engine visualization

`on` (default) | `off`

Select whether to run simulations in the 3D visualization environment without visualizing the results, that is, in headless mode.

Consider running in headless mode in these cases:

- You want to run multiple 3D simulations in parallel to test models in different Unreal Engine scenarios.

Dependencies

To enable this parameter, set **Scene source** to `Default Scenes` or `Unreal Executable`.

Weather

Override scene weather — Option to control the scene weather and sun position

off (default) | on

Select whether to control the scene weather and sun position during simulation. Use the enabled parameters to change the sun position, clouds, fog, rain, and snow.

This table summarizes sun position settings for specific times of day.

Time of Day	Settings	Unreal Editor Environment
Midnight	**Sun altitude**: -90 **Sun azimuth**: 180	
Sunrise in the north	**Sun altitude**: 0 **Sun azimuth**: 180	
Noon	**Sun altitude**: 90 **Sun azimuth**: 180	

This table summarizes settings for specific cloud conditions.

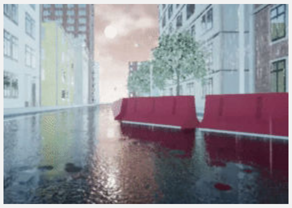
Cloud Condition	Settings	Unreal Editor Environment
Clear	**Cloud opacity**: 0	
Heavy	**Cloud opacity**: 85	
Heavy	**Enable Volumetric clouds**: Selected **Cloud Coverage**: 50 **Cloud layer altitude**: 6	

This table summarizes settings for specific fog conditions.

Fog Condition	Settings	Unreal Editor Environment
None	**Fog density**: 0	

Fog Condition	Settings	Unreal Editor Environment
Heavy	**Fog density**: 100	

This table summarizes settings for specific rain conditions.

Rain Condition	Settings	Unreal Editor Environment
Light	**Cloud opacity**: 10 **Rain density**: 25	
Heavy	**Cloud opacity**: 10 **Rain density**: 80	

This table summarizes settings for specific snow conditions.

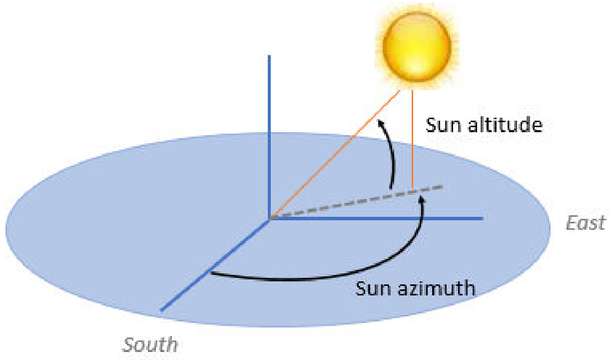
Snow Condition	Settings	Unreal Editor Environment
Heavy	**Snow density**: 50	

Sun Settings

Sun azimuth — Azimuth angle from south to horizontal projection of the sun ray

`90` (default) | any value between `0` and `360`

Azimuth angle in the horizontal plane measured from the south to the horizontal projection of the sun rays, in deg.



Use the **Sun altitude** and **Sun azimuth** parameters to control the time of day in the scene. For example, to specify sunrise in the north, set **Sun altitude** to 0 deg and **Sun azimuth** to 180 deg.

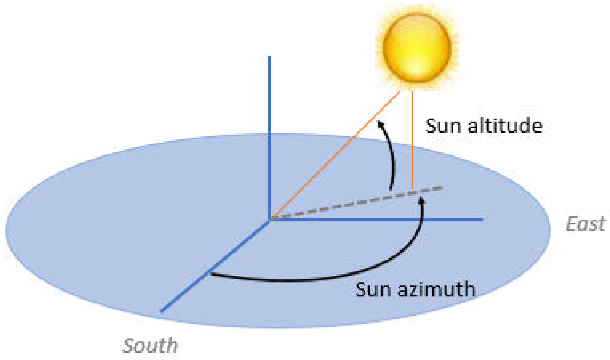
Dependencies

To enable this parameter, select **Override scene weather**.

Sun altitude — Altitude angle between sun and horizon

`40` (default) | any value between `-90` and `90`

Altitude angle in a vertical plane between the sun's rays and the horizontal projection of the rays, in deg.



Use the **Sun altitude** and **Sun azimuth** parameters to control the time of day in the scene. For example, to specify sunrise in the north, set **Sun altitude** to 0 deg and **Sun azimuth** to 180 deg.

Dependencies

To enable this parameter, select **Override scene weather**.

Enable geospatial sun — Option to enable geospatial sun
`off` (default) | `on`

Select this check box to enable geospatial sun. Enabling geospatial sun is useful to simulate conditions near the poles or at locations with disproportionate lengths of day versus night.

Dependencies

To enable this parameter, select **Override scene weather**.

Latitude — Latitude of geolocation on earth

`0` (default) | scalar

Latitude of geolocation on earth, specified as a scalar, in deg.

Dependencies

To enable this parameter, select **Override scene weather** and **Enable geospatial sun**.

Longitude — Longitude of geolocation on earth

0 (default) | scalar

Longitude of geolocation on earth, specified as a scalar, in deg.

Dependencies

To enable this parameter, select **Override scene weather** and **Enable geospatial sun**.

Date (YYYY-MM-DD) — Date

2023-01-01 (default) | integer

Date, specified as an integer in the format `YYYY-MM-DD`.

Dependencies

To enable this parameter, select **Override scene weather** and **Enable geospatial sun**.

Time (HH:MM:SS) — Time

`10:30:00` (default) | integer

Time, specified as an integer in the format `HH:MM:SS`.

Dependencies

To enable this parameter, select **Override scene weather** and **Enable geospatial sun**.

Time zone — Time zone

`-5.0` (default) | any value between `-12.0` and `12.0`

Time zone specifies the number of hours offset from the Coordinated Universal Time (UTC) or Greenwich Mean Time (GMT).

Dependencies

To enable this parameter, select **Override scene weather** and **Enable geospatial sun**.

Daylight saving time — Option to enable daylight saving time

`off` (default) | `on`

Select this check box to enable daylight saving time.

Dependencies

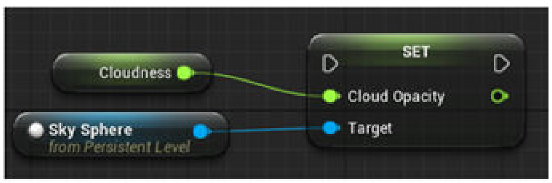
To enable this parameter, select **Override scene weather** and **Enable geospatial sun**.

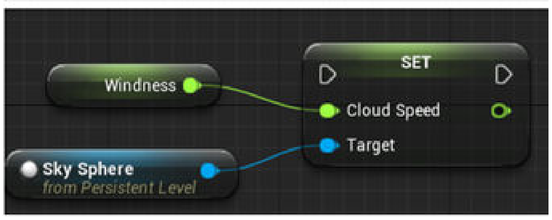
Cloud Settings

Cloud speed — Unreal Editor Cloud Speed global actor target value

`1` (default) | any value between `-100` and `100`

Parameter that corresponds to the Unreal Editor **Cloud Speed** global actor target value. The clouds move from west to east for positive values and east to west for negative values.





Use the **Cloud opacity** and **Cloud speed** parameters to control clouds in the scene.

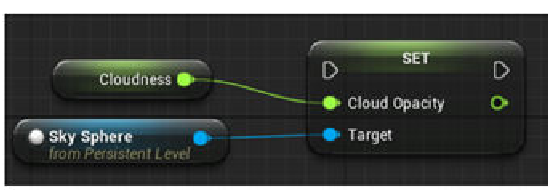
Dependencies

To enable this parameter, select **Override scene weather**.

Cloud opacity — Unreal Editor Cloud Opacity global actor target value

`10` (default) | any value between `0` and `100`

Parameter that corresponds to the Unreal Editor **Cloud Opacity** global actor target value, in percent. Zero is a cloudless scene.



Use the **Cloud opacity** and **Cloud speed** parameters to control clouds in the scene.

Dependencies

To enable this parameter, select **Override scene weather**.

Enable volumetric clouds — Option to enable volumetric clouds
`off` (default) | `on`

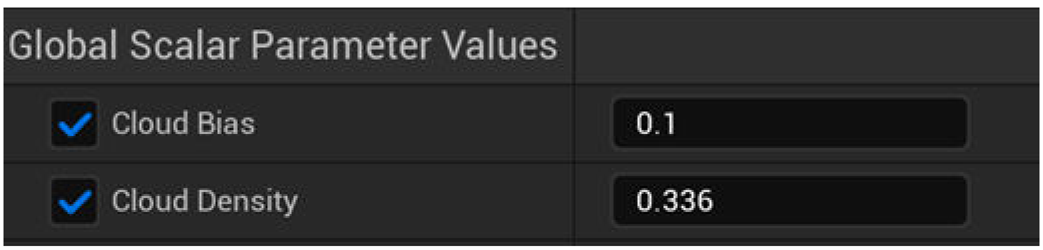
Select this check box to enable volumetric clouds.

Dependencies

To enable this parameter, select **Override scene weather**.

Cloud coverage — Unreal Editor Cloud Density global target value
`1.0` (default) | any value between `0` and `100`

Parameter that corresponds to the Unreal Editor **Cloud Density** global actor target value, in percent.



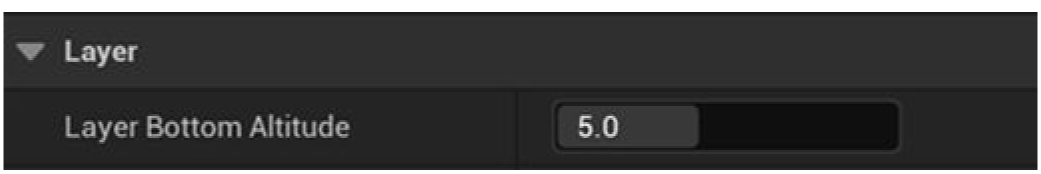
Use the **Cloud coverage** parameter to control clouds in the scene.

Dependencies

To enable this parameter, select **Override scene weather** and **Enable volumetric clouds**.

Cloud layer altitude — Unreal Editor Layer Bottom Altitude global actor target value
`6` (default) | any value between `5` and `20`

Parameter that corresponds to the Unreal Editor **Layer Bottom Altitude** global actor target value, in km.



Dependencies

To enable this parameter, select **Override scene weather** and **Enable volumetric clouds**.

Fog Settings

Fog density — Unreal Editor Set Fog Density and Set Start Distance target values

`0` (default) | any value between `0` and `100`

Parameter that corresponds to the Unreal Editor **Set Fog Density** and **Set Start Distance** target values, in percent.

Dependencies

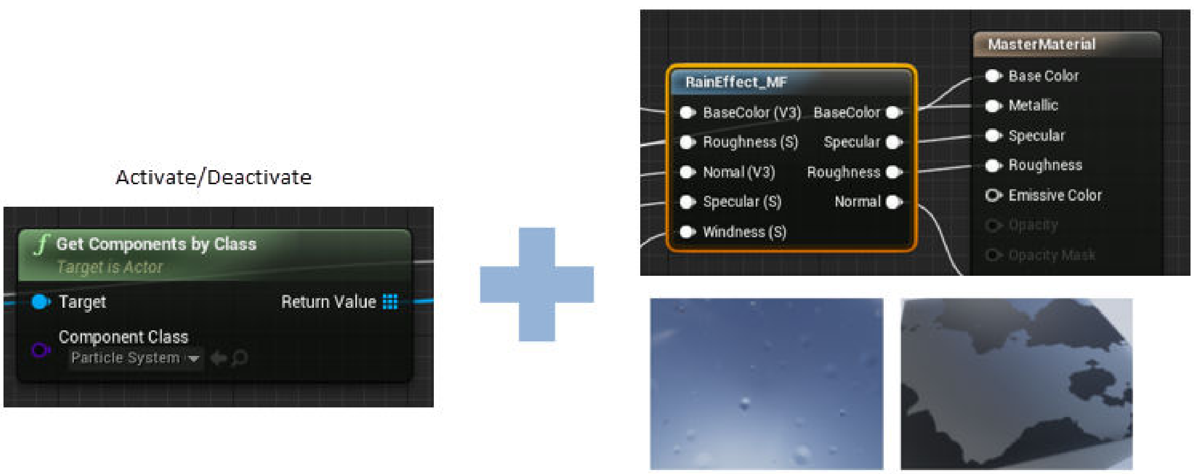
To enable this parameter, select **Override scene weather**.

Rain Settings

Rain density — Unreal Editor local actor controlling rain density, wetness, rain puddles, and ripples

`0` (default) | any value between `0` and `100`

Parameter corresponding to the Unreal Editor local actor that controls rain density, wetness, rain puddles, and ripples, in percent.



Wetness, rain puddles, and ripples

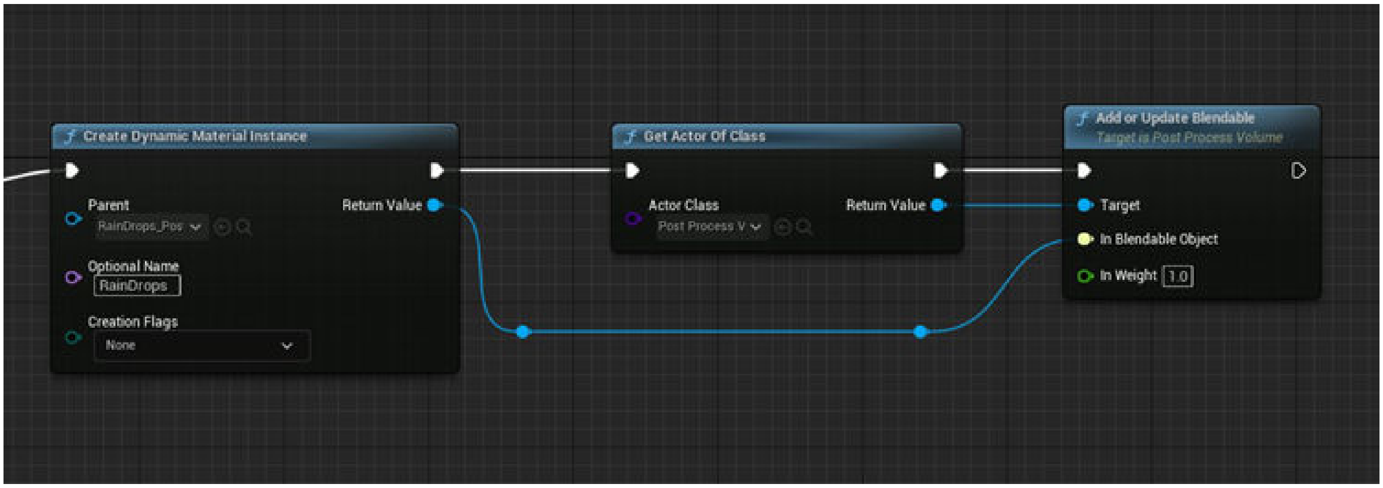
Use the **Cloud opacity** and **Rain density** parameters to control rain in the scene.

Dependencies

To enable this parameter, select **Override scene weather**.

Enable raindrops on camera lens — Option to enable raindrops on camera lens
`off` (default) | `on`

Select this check box to enable raindrops on the camera lens.



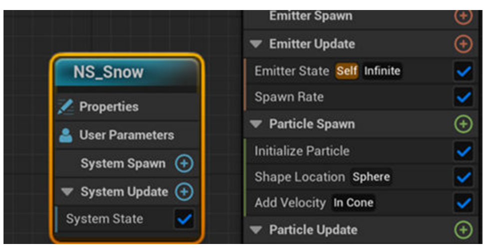
Dependencies

To enable this parameter, select **Override scene weather**.

Snow Settings

Snow density — Unreal Editor global actor controlling snow density
`0` (default) | any value between `0` and `100`

This parameter corresponds to the Unreal Editor global actor that controls snow density, in percent.



Dependencies

To enable this parameter, select **Override scene weather**.

Geospatial

Enable geospatial configuration — Option to enable geospatial parameters and variant subsystem

`off` (default) | `on`

Select this check box to enable geospatial parameters and a variant subsystem.

Access token ID — ID of stored token

string representing ID of stored token

ID of stored token, specified as a string. To create this token, create a Cesium Ion account, then generate the token through this account. The Authentication Manager stores the token from Cesium Ion. For more information, see "Visualize with Cesium" on page 2-42 and https://ion.cesium.com/.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** check box.

Use Earth center as origin (ECEF) — Use Earth center as origin (ECEF)

`off` (default) | on

Select this check box to position aerospace vehicles with respect to the Earth center. Use this option with the ECEF coordinate frame selection in the pack blocks.

Origin height (m) — Height at georeference point on globe

real scalar

Height at georeference point on the globe, specified as a real scalar. This parameter represents the height above the 1984 World Geodetic System (WGS84) ellipsoid model of the Earth at the latitude and longitude specified in **Origin latitude** and **Origin longitude**.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Clear the **Use Earth center as origin (ECEF)** check box.

Origin latitude — Latitude

real scalar

Latitude, specified as a real scalar in decimal degrees.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Clear the **Use Earth center as origin (ECEF)** check box.

Origin longitude — Longitude

real scalar

Longitude, specified as a real scalar in decimal degrees.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Clear the **Use Earth center as origin (ECEF)** check box.

Map style — Raster overlay type

`Aerial` (default) | `Aerial with labels` | `Road`

Raster overlay type, specified as `Aerial`, `Aerial with labels`, or `Road`.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** check box.

Additional asset IDs — Local dataset IDs

`[]` (default) | array | vector

Local dataset IDs, specified as an array or vector.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** check box.

Use advanced Sun sky — Georeferenced, location-accurate Sun Sky actor

`off` (default) | on

Select this check box to add a georeferenced, location-accurate Sun Sky actor in simulation.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** check box.

Solar time — Current solar time

`11` (default) | scalar real

Current solar time, specified as scalar hours from midnight.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

Time zone — Time zone

`11` (default) | scalar real

Time zone, specified as hours offset from Greenwich Mean Time (GMT). To specify hours before GMT, use a minus sign (`-`).

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

Day — Day

`21` (default) | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `10` | `11` | `12` | `13` | `14` | `15` | `16` | `17` | `18` | `19` | `20` | `21` | `22` | `23` | `24` | `25` | `26` | `27` | `28` | `29` | `30` | `31`

Day, specified as a scalar from 1 to 31.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

Month — Month

9 (default) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12

Month, specified as a scalar from 1 to 12.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

Year — Year

2022 (default) | scalar real

Year, specified as a scalar real.

Use daylight saving time (DST) — Daylight saving time

off (default) | on

Select this check box to enable daylight saving time.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

DST start day — Start day of daylight saving time

10 (default) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31

Start day of daylight saving time, specified as a scalar from 1 to 31.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

DST start month — Start month of daylight saving time

3 (default) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12

Start month of daylight saving time, specified as a scalar from 1 to 12.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

DST end day — Last day of daylight saving time

3 (default) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31

Last day of daylight saving time, specified as a scalar from 1 to 31.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

DST end month — Last month of daylight saving time

11 (default) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12

Last month of daylight saving time, specified as a scalar from 1 to 12.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

DST switch hour — Hour when daylight saving time switches

2 (default) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24

Hour when daylight saving time switches, specified as a scalar from 1 to 24.

Dependencies

To enable this parameter:

- Select the **Enable geospatial configuration** check box.
- Select the **Use advanced Sun sky** check box.

Authentication manager — Management of access tokens
button

Click to manage access tokens, such as create, update, and delete tokens.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** check box.

More About

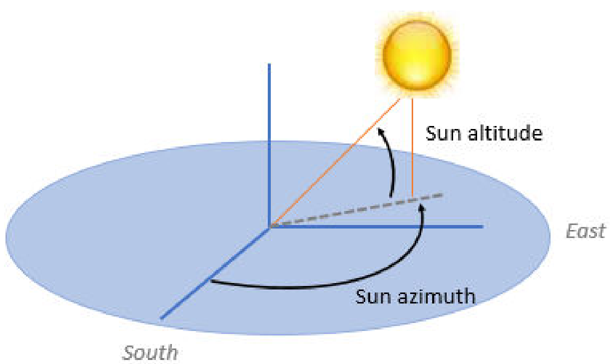
Sun Position and Weather

To control the scene weather and sun position, on the **Weather** tab, select **Override scene weather**. Use the enabled parameters to change the sun position, clouds, fog, and rain during the simulation.

Sun Position

Use **Sun altitude** and **Sun azimuth** to control the sun position.

- **Sun altitude** — Altitude angle in a vertical plane between the sun rays and the horizontal projection of the rays.
- **Sun azimuth** — Azimuth angle in the horizontal plane measured from the south to the horizontal projection of the sun rays.



Alternatively, select **Enable geospatial sun** to use **Latitude**, **Longitude**, **Date**, **Time**, and **Time zone** to control sun position. Enabling geospatial sun is useful to simulate conditions near the poles or at locations with disproportionate lengths of day versus night.

This table summarizes sun position settings for specific times of day.

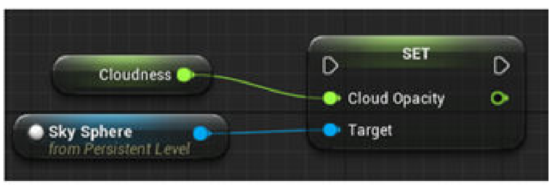
Time of Day	Settings	Unreal Editor Environment
Midnight	**Sun altitude**: -90 **Sun azimuth**: 180	

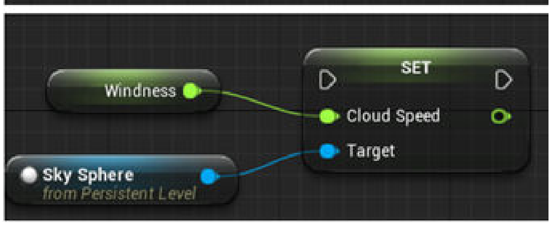
Time of Day	Settings	Unreal Editor Environment
Sunrise in the north	**Sun altitude**: 0 **Sun azimuth**: 180	
Noon	**Sun altitude**: 90 **Sun azimuth**: 180	

Clouds

Use **Cloud opacity** and **Cloud speed** to control clouds in the scene.

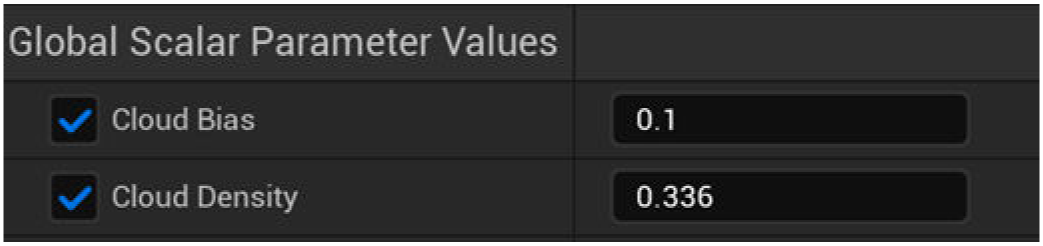
- **Cloud opacity** — Unreal Editor **Cloud Opacity** global actor target value. Zero is a cloudless scene.
- **Cloud speed** — Unreal Editor **Cloud Speed** global actor target value. The clouds move from west to east for positive values and east to west for negative values.



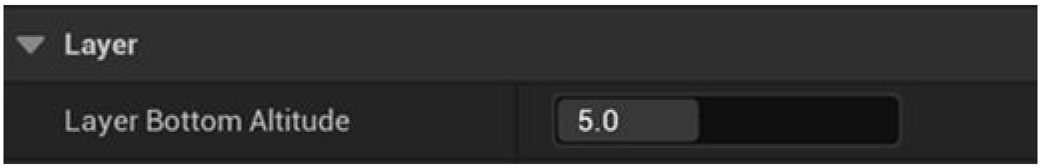


Select **Enable volumetric clouds** to use **Cloud coverage** and **Cloud layer altitude** to control volumetric clouds in the scene.

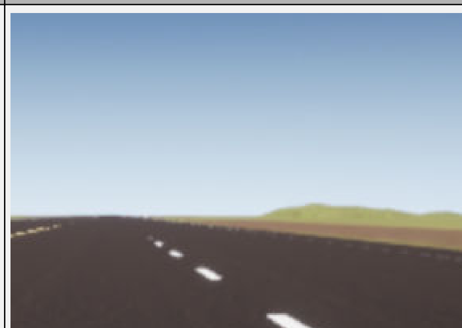
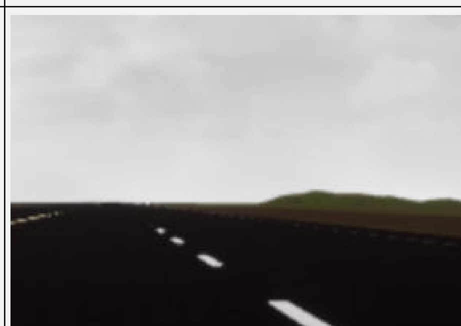
- **Cloud coverage** — Unreal Editor **Cloud Density** global actor target value.



- **Cloud layer altitude** — Unreal Editor **Layer Bottom Altitude** global actor target value.



This table summarizes settings for specific cloud conditions.

Cloud Condition	Settings	Unreal Editor Environment
Clear	**Cloud opacity**: 0	
Heavy	**Cloud opacity**: 85	

Cloud Condition	Settings	Unreal Editor Environment
Heavy	**Enable Volumetric clouds**: Selected **Cloud Coverage**: 50 **Cloud layer altitude**: 6	

Fog

Use **Fog density** to control fog in the scene. **Fog density** corresponds to the Unreal Editor **Set Fog Density**.

This table summarizes settings for specific fog conditions.

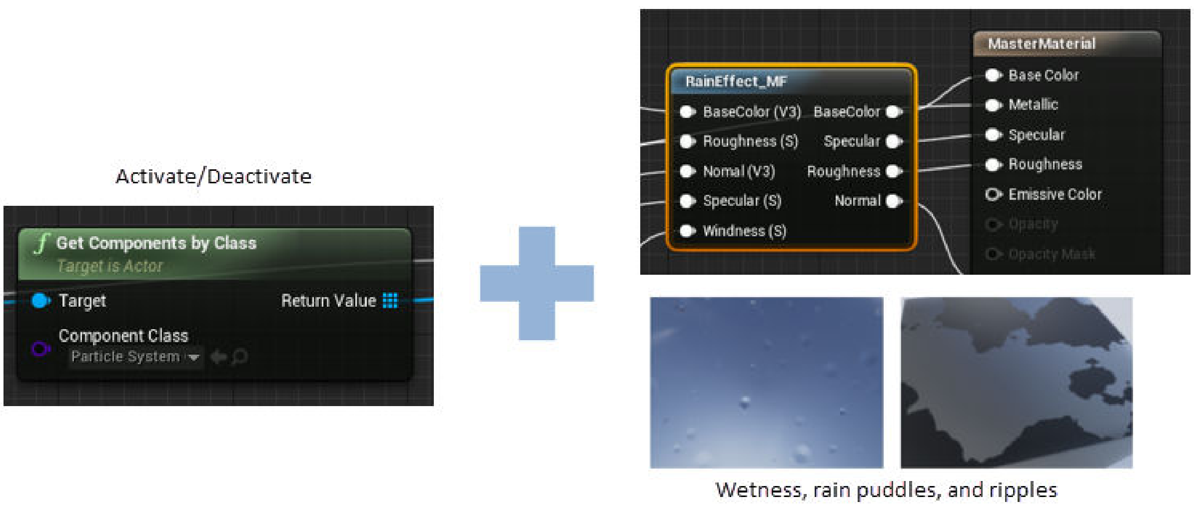
Fog Condition	Settings	Unreal Editor Environment
None	**Fog density**: 0	

Fog Condition	Settings	Unreal Editor Environment
Heavy	**Fog density**: 100	

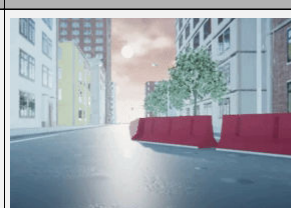
Rain

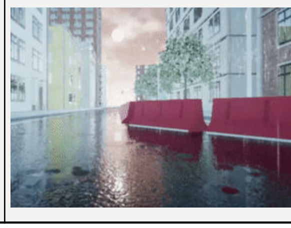
Use **Cloud opacity** and **Rain density** to control rain in the scene.

- **Cloud opacity** — Unreal Editor **Cloud Opacity** global actor target value.
- **Rain density** — Unreal Editor local actor that controls rain density, wetness, rain puddles, and ripples.



This table summarizes settings for specific rain conditions.

Rain Condition	Settings	Unreal Editor Environment
Light	**Cloud opacity**: 10 **Rain density**: 25	

Rain Condition	Settings	Unreal Editor Environment
Heavy	**Cloud opacity**: 10 **Rain density**: 80	

Snow

Use **Snow density** to control snow in the scene.

- **Snow density** — Unreal Editor global actor that controls snow density.

This table summarizes settings for specific snow conditions.

Snow Condition	Settings	Unreal Editor Environment
Heavy	**Snow density**: 50	

Version History

Introduced in R2021b

R2023b: Import RoadRunner Scene into Unreal Engine Simulation Environment

Use the Simulation 3D Scene Configuration block to import a RoadRunner scene and simulate in the scene. To specify the source of scene, select the **Scene source** parameter as `RoadRunner`. Then, in the **Project** parameter, specify the Filmbox `(.fbx)` file corresponding to the RoadRunner scene. For an example, see “Import RoadRunner Scene into Unreal Engine Using Simulink” (Simulink 3D Animation).

R2023b: Camera View Improvements to View the Virtual World

Use the Simulation 3D Scene Configuration block to view the virtual world from a camera positioned at a custom viewpoint. To set the custom viewpoint, select **Scene view** as `Custom` and set the **Initial viewer translation** and **Initial viewer rotation** values. You can also use keyboard shortcuts and mouse controls to navigate in the virtual world scene.

R2024a: Requires Simulink 3D Animation
Behavior change in future release

Simulating models with the Simulation 3D Scene Configuration block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Scene Configuration to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

R2024a: New parameters available to simulate additional weather conditions

Use new weather parameters to simulate additional weather conditions.

Weather	Parameters
Sun Settings	• **Enable geospatial sun** • **Latitude** • **Longitude** • **Date** • **Time** • **Daylight saving time**
Cloud Settings	• **Enable volumetric clouds** • **Cloud coverage** • **Cloud layer altitude**
Rain Settings	• **Enable raindrops on camera lens**
Snow Settings	• **Snow density**

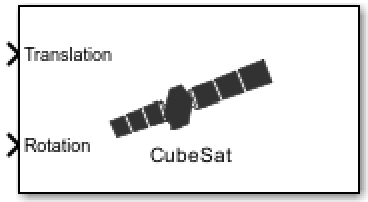
See Also

Topics

"Visualize with Cesium" on page 2-42
"How 3D Simulation for Aerospace Blockset Works" on page 2-40
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"Prepare Custom Aircraft Mesh for the Unreal Editor" on page 4-34

Simulation 3D Spacecraft

Implement spacecraft in 3D environment



Libraries:
Aerospace Blockset / Animation / Simulation 3D

Description

Note Simulating models with the Simulation 3D Spacecraft block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Spacecraft to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Spacecraft block implements spacecraft in a 3D visualization environment using translation and rotation to place the spacecraft.

To use this block, your model must contain a Simulation 3D Scene Configuration block. If you set the **Sample time** parameter of the Simulation 3D Spacecraft block to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

The block input uses the spacecraft north-east-down (NED) *right-handed* (RH) *Cartesian* coordinate system.

- *X*-axis — Along spacecraft longitudinal axis, points forward
- *Y*-axis — Along spacecraft lateral axis, points to the right
- *Z*-axis — Points downward

For more information, see “About Aerospace Coordinate Systems” on page 2-8.

Tip Verify that the Simulation 3D Spacecraft block executes before the Simulation 3D Scene Configuration block. That way, the Simulation 3D Spacecraft block prepares the signal data before the Unreal Engine 3D visualization environment receives the data. To check the block execution order, right-click each block and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Spacecraft — `-1`

For more information about execution order, see “Control and Display Execution Order”.

Skeletons, Bones, and Meshes

Unreal Engine uses a skeleton, bones, and a mesh to define a 3D model. A skeleton is comprised of a set of bones. A mesh is the outer covering of the skeleton. Spacecraft parts are sections of the mesh, such as antenna or thrusters, that are linked to the bones. For more information, see https://docs.unrealengine.com/4.27/AnimatingObjects/SkeletalMeshAnimation/Skeleton/.

For more information on how the Simulation 3D Space block translation input arrays connect to spacecraft types, see “Algorithms” on page 5-1130.

Ports

Input

Translation — Spacecraft translation
10-by-3 array | 20-by-3 array

Spacecraft translation, specified as:

- 10-by-3 array — Spacecraft **Type** is `CubeSat`.
- 20-by-3 array — Spacecraft **Type** is `SmallSat`.

The signal contains translation [*X*, *Y*, *Z*], in meters, with one row of the array for each bone of the spacecraft.

The translation applies to these bones of the `CubeSat` type.

Bone	Index
BODY	1
ANTENNA1	2
ANTENNA2	3
ANTENNA3	4
SENSOR	5
SOLAR_ARRAY1	6
SOLAR_ARRAY1_1	7
SOLAR_ARRAY2	8
SOLAR_ARRAY2_2	9
THRUSTER	10

The translation applies to these bones of the `SmallSat` type.

Bone	Index
BODY	1
ANTENNA1	2
ANTENNA2	3
ANTENNA3	4

Bone	Index
ANTENNA4	5
SENSOR1	6
SENSOR2	7
SENSOR3	8
SENSOR4	9
SOLAR_ARRAY1	10
SOLAR_ARRAY1_1	11
SOLAR_ARRAY1_1_1	12
SOLAR_ARRAY1_1_1_1	13
SOLAR_ARRAY1_1_1_1_1	14
SOLAR_ARRAY2	15
SOLAR_ARRAY2_2	16
SOLAR_ARRAY2_2_2	17
SOLAR_ARRAY2_2_2_2	18
SOLAR_ARRAY2_2_2_2_2	19
THRUSTER	20

Data Types: `single` | `double`

Rotation — Spacecraft rotation
10-by-3 array | 20-by-3 array

Spacecraft rotation, specified as:

- 10-by-3 array — Spacecraft **Type** is `CubeSat`.
- 20-by-3 array — Spacecraft **Type** is `SmallSat`.

The rotation applies to the same bones as listed for the "Translation" on page 5-0 port.

The signal contains the rotation [roll, pitch, yaw], in radians, with one row of the array for each bone of the spacecraft.

Data Types: `single` | `double`

Parameters

To edit block parameters interactively, use the Property Inspector. From the Simulink Toolstrip, on the **Simulation** tab, in the **Prepare** gallery, select **Property Inspector**.

Type — Spacecraft type

`CubeSat` (default) | `SmallSat`

Spacecraft type, specified as `CubeSat` or `SmallSat`.

Dependencies

To set this parameter, set the **Initial translation** and **Initial rotation** parameters to the matching array size. Failure to appropriately set these array sizes causes an error.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Mesh`
Values:	`CubeSat` (default) | `SmallSat`

Example: `CubeSat`

Path to CubeSat mesh — Path to CubeSat mesh

`/MathWorksAerospaceContent/Vehicles/Spacecraft/CubeSat/Mesh/SK_CubeSat.SK_CubeSat` (default) | character vector

Path to CubeSat mesh, specified as a character vector.

Dependencies

To enable this parameter, set **Type** to `CubeSat`.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`MeshPath`
Values:	`/MathWorksAerospaceContent/Vehicles/Spacecraft/CubeSat/Mesh/SK_CubeSat.SK_CubeSat` | character vector

Example: `/MathWorksAerospaceContent/Vehicles/Spacecraft/CubeSat/Mesh/SK_CubeSat.SK_CubeSat`

Path to SmallSat mesh — Path to SmallSat mesh

`/MathWorksAerospaceContent/Vehicles/Spacecraft/SmallSat/Mesh/SK_SmallSat.SK_SmallSat` (default) | character vector

Path to SmallSat mesh, specified as a character vector.

Dependencies

To enable this parameter, set **Type** to `SmallSat`.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`MeshPathSmall`

Values:	`/MathWorksAerospaceContent/Vehicles/Spacecraft/SmallSat/Mesh/SK_SmallSat.SK_SmallSat` | character vector

Example: `/MathWorksAerospaceContent/Vehicles/Spacecraft/SmallSat/Mesh/SK_SmallSat.SK_SmallSat`

Name — Spacecraft name

`SimulinkVehicle1` (default) | character vector

Spacecraft name, specified as a character vector. By default, when you use the Simulation 3D Spacecraft block in your model, the block sets the **Name** parameter to `SimulinkVehicle`*X*. The value of *X* depends on the number of other Simulation 3D blocks that you have in your model.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`ActorName`
Values:	`SimulinkVehicle1` (default) | character vector

Example: `SimulinkVehicle1`

Initial translation (m) — Initial translation of spacecraft

`zeros(10,3)` (default) | 10-by-3 array

Initial translation of spacecraft, specified as a 10-by-3 array.

Dependencies

This parameter must match the spacecraft type you set in **Type**. Failure to appropriately set these array sizes causes an error.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Translation`
Values:	`zeros(10,3)` (default) | 10-by-3 array

Example: `zeros(10,3)`

Data Types: `single` | `double`

Initial rotation (rad) — Initial rotation of spacecraft

`zeros(10,3)` (default) | 10-by-3 array

Initial rotation of spacecraft, specified as a 10-by-3 array.

Dependencies

This parameter must match the spacecraft type you set in **Type**. Failure to appropriately set these array sizes causes an error.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`Rotation`
Values:	`zeros(10,3)` (default) | 10-by-3 array

Example: `zeros(10,3)`

Data Types: `single` | `double`

Sample time — Sample time

`0.02` (default) | real scalar

Sample time, T_s. The graphics frame rate is the inverse of the sample time.

Programmatic Use

To set the block parameter value programmatically, use the `set_param` function.

To get the block parameter value programmatically, use the `get_param` function.

Parameter:	`SampleTime`
Values:	`'-1'` (default) | real scalar

Example: `'-1'`

Algorithms

CubeSat and SmallSat Spacecraft Types

For CubeSat and SmallSat spacecraft types, the Simulation 3D Spacecraft block enables six degrees of freedom (6DOF) for all spacecraft bones. Note, the default poses for each bone in the skeletal mesh affects how the mesh deforms when you manipulate each degree of freedom.

Version History

Introduced in R2024a

See Also

Blocks
Simulation 3D Scene Configuration | Simulation 3D Actor Transform Get | Simulation 3D Actor Transform Set | Simulation 3D Camera Get | Simulation 3D Message Get | Simulation 3D Message Set

Tools
CubeSat | Simulation 3D CubeSat Pack | SmallSat | Simulation 3D SmallSat Pack

Simulation Pace

Set simulation rate for animation viewing

Libraries:
Aerospace Blockset / Animation / Animation Support Utilities

Description

The Simulation Pace block lets you run model simulation at a slower pace so that you can comfortably view connected animations and understand and observe the system behavior. Visualizing simulations at a slower rate makes it easier to understand underlying system design, identify design issues and demonstrate near real-time behavior. You can view the results and inspect your system while the simulation is in progress.

Use this block in scenarios where one simulation-second is completed in a few wall clock time milliseconds.

When configuring this block, also consider the block sample time, which affects the simulation pace. The default is 1/30th of a second, which corresponds to a 30 frames-per-second visualization rate (typical for desktop computers). For more information, see “Sample time” on page 5-0 .

To use this block:

- Set the model solver to `Fixed-step`.
- Use a discrete sample time.

Tip The Simulation Pace block:

- Does not produce deployable code.
- Is not supported in referenced models for simulation in accelerator or rapid accelerator mode. To slow down the simulation in these modes, in the **Simulation** tab of the Simulink Editor toolstrip, select **Run > Simulation Pacing**.

Ports

Output

Port_1 — Pace error
scalar

Pace error, specified as a scalar.

The block optionally outputs the pace error value (*simulationTime* minus *ClockTime*), in seconds. The pace error is positive if the simulation is running faster than the specified pace and negative if slower than the specified pace.

Outputting the pace error from the block lets you record the overall pace achieved during the simulation or routing the signal to other blocks to determine if the simulation is too slow to keep up with the specified pace.

Dependencies

To enable this port, select the **Output pace error (sec)** check box.

Data Types: `double`

Parameters

Simulation pace — Ratio of simulation time to clock time

`1` (default) | scalar

Ratio of simulation time to clock time, specified as a scalar, in seconds of simulation time per second of clock time.

Programmatic Use
Block Parameter: `OutputPaceError`
Type: character vector
Values:`'1'` | scalar
Default: `'1'`

Sleep mode — Control simulation pace

`Auto` (default) | `MATLAB Thread` | `Off` | `Busy-Wait`

Control simulation pace of model using one of these methods. `MATLAB Thread`, `Busy-Wait`, and `Auto` slow down the simulation pace at simulation-second 0.1 to wait for the wall clock to get to time 1. Use this parameter when one simulation-second is completed in a few wall clock time milliseconds.

- `Auto` — Use the model configuration parameter setting **Enable pacing to slow down simulation** to control the simulation pace. If the model configuration parameter setting **Enable pacing to slow down simulation** is not selected, the block behaves as though the `MATLAB Thread` option is selected.
- `MATLAB Thread` — Use the operating system `sleep` function during simulation to wait for the wall clock to get to time 1.
- `Off` — Disable the pace functionality and let the simulation run as fast as possible.
- `Busy-Wait` — Use a while loop in conjunction with the Simstruct to wait for the simulation to wait for the wall clock to get to time 1.

Programmatic Use
Block Parameter: `SleepMode`
Type: character vector
Values:`'MATLAB Thread'` | `'Off'` | `'Busy-Wait'` | `'Auto'`
Default: `'Auto'`

Output pace error — Display pace error

`off` (default) | `on`

Select this check box to output the pace error value (*simulationTime* minus *ClockTime*), in seconds. The pace error is positive if the simulation is running faster than the specified pace and negative if slower than the specified pace. To disable the display, clear this check box .

Programmatic Use
Block Parameter: `OutputPaceError`
Type: character vector
Values:`'off' | 'on'`
Default: `'off'`

Sample time — Sample time

1/30 (default) | `-1` | scalar | vector

Specify the sample time as a scalar. The default 1/30th of a second corresponds to a 30 frames-per-second visualization rate (typical for desktop computers). To set how often the Simulink interface synchronizes with the wall clock, use this parameter.

The block sample time must be:

- Discrete
- Greater than 0.0 or an inherited sample time (-1)

The block sample time and its optional offset time ($[T_s,\ T_o]$) must be finite and discrete.

Caution Choose as slow a sample time as needed for smooth animation, since oversampling has little benefit and undersampling can cause animation jumpiness. Undersampling can also potentially block the MATLAB main thread on your computer.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'1/30'`

Algorithms

The simulation pace is implemented by putting the entire MATLAB thread to sleep until it must run again to keep up the pace. The Simulink software is single threaded and runs on the one MATLAB thread, so only one Simulation Pace block can be active at a time.

Version History

Introduced before R2006a

R2023a: Simulation Pace Block Support for Referenced Models

Behavior changed in R2023a

The Simulation Pace block is not supported in referenced models for simulation in accelerator or rapid accelerator mode. Starting in R2023a, to slow down the simulation in these modes, in the **Simulation** tab of the Simulink Editor toolstrip, select **Run > Simulation Pacing**.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Pilot Joystick

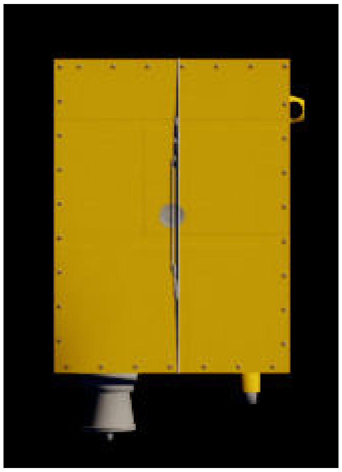
Tools
Simulation Pacing Options

Topics
"Specify Sample Time"
"Quadcopter Modeling and Simulation based on Parrot Minidrone" on page 9-81

SmallSat

Generic SmallSat

Description



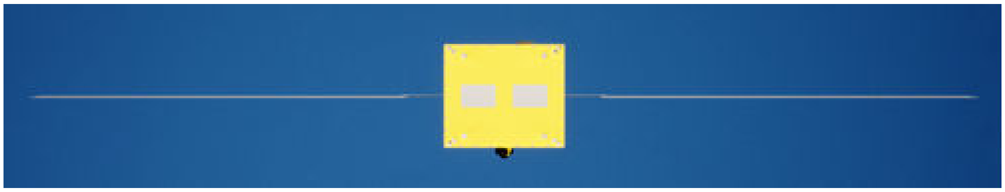
SmallSat is one of the spacecraft that you can use within the 3D simulation environment. The SmallSat represents a small satellite, typically under 500 kg. This environment is rendered using the Unreal Engine from Epic Games. For detailed views of the SmallSat, see “Views” on page 5-477.

To add this type of vehicle to the 3D simulation environment:

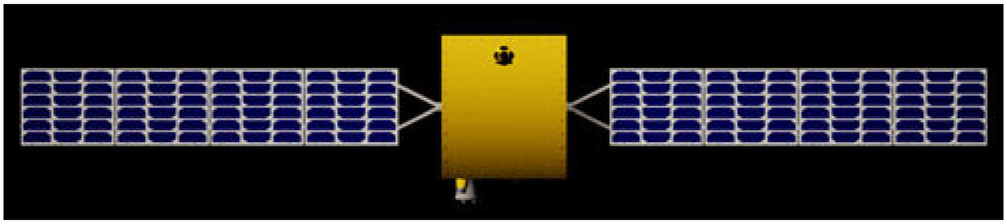
1. Add a Simulation 3D Spacecraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Parameters** tab, set the **Type** parameter to `SmallSat`.
3. Set the **Initial translation (m)** and **Initial rotation (rad)** parameters to an array size that matches SmallSat, for example, `zeros(20,3)`.

Views

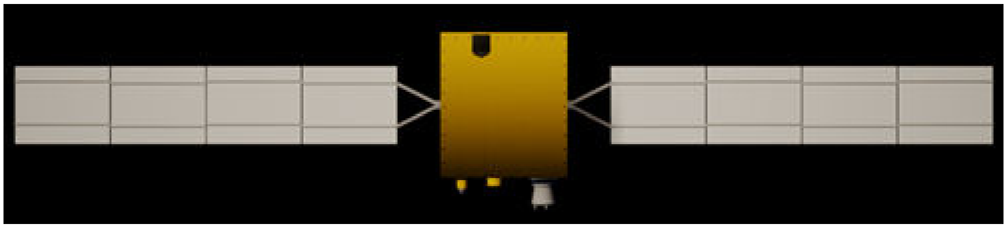
`Top-down view` — SmallSat top-down view
diagram



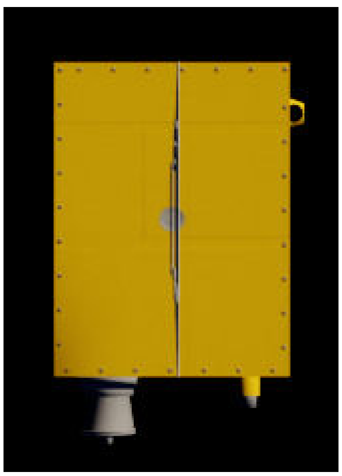
`Left side view` — SmallSat left side view
diagram



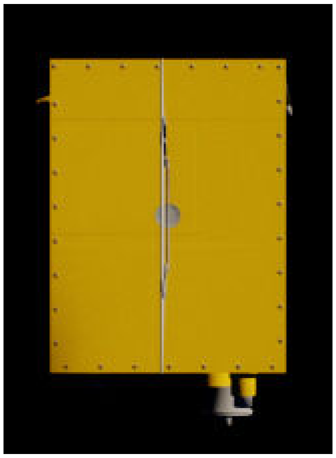
Right side view — SmallSat right side view
diagram



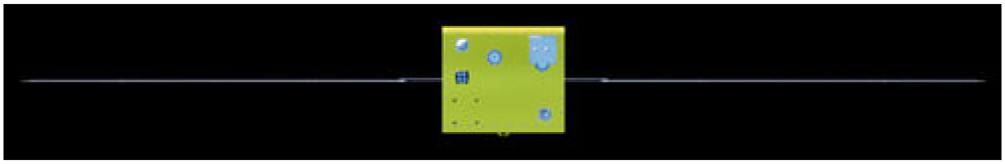
Front view — SmallSat front view
diagram



Back view — SmallSat back view
diagram



`Bottom view` — SmallSat bottom view diagram



Skeleton

SmallSat Skeleton

The SmallSat skeleton has these parts.

- `SmallSat`
  - `ANTENNA1`
  - `ANTENNA2`
  - `ANTENNA3`
  - `ANTENNA4`
  - `SENSOR1`
  - `SENSOR2`
  - `SENSOR3`
  - `SENSOR4`
  - `SOLAR_ARRAY1`
    - `SOLAR_ARRAY1_1`
      - `SOLAR_ARRAY1_1_1`
        - `SOLAR_ARRAY1_1_1_1`
          - `SOLAR_ARRAY1_1_1_1_1`
  - `SOLAR_ARRAY2`

        - SOLAR_ARRAY2_2
            - SOLAR_ARRAY2_2_2
                - SOLAR_ARRAY2_2_2_2
                    - SOLAR_ARRAY2_2_2_2_2
- THRUSTER

Version History

Introduced in R2024a

See Also

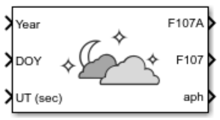
Blocks
Simulation 3D Spacecraft

Tools
CubeSat

Topics
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Solar Flux and Geomagnetic Index

Return solar flux and geomagnetic index



Libraries:
Aerospace Blockset / Environment / Celestial Phenomena

Description

The Solar Flux and Geomagnetic Index block extracts the solar flux and geomagnetic index from the MAT-file that is generated as `aeroSpaceWeatherData.mat` by the `aeroReadSpaceWeatherData` function.

Limitations

When specifying **Magnetic index extrapolation value** as `IGRF`, the Solar Flux and Geomagnetic Index block predicts the elements of the magnetic index information outside the MAT-file data range using the International Geomagnetic Reference Field. Because this model is defined for times between January 1, 1900, 12:00 AM UTC and January 1, 2025, 12:00 AM UTC, the predictions for times outside this range are clipped to values at these time limits.

Ports

Input

Year — Year
scalar

Year, specified as a scalar. The `aeroReadSpaceWeatherData` function ignores the value of `year`.

Dependencies

To enable this port, set the **Input time specification** parameter to `Year, day, sec`.

Data Types: `double`

DOY — Day of year
scalar

Day of year, specified as a scalar.

Dependencies

To enable this port, set the **Input time specification** parameter to `Year, day, sec`.

Data Types: `double`

UT (sec) — Universal time
scalar

Universal time (UT), specified as a scalar in seconds. This value is the number of seconds relative to midnight of **DOY**.

Dependencies

To enable this port, set the **Input time specification** parameter to `Year, day, sec`.

Data Types: `double`

T_{JD} — Universal time
scalar

Universal time (UT), specified as a scalar Julian date.

Dependencies

To enable this port, set the **Input time specification** parameter to `Julian date`.

Data Types: `double`

Output

F107A — 81-day average of F10.7 flux
scalar

81-day average of F10.7 flux, centered on day of year (`dayOfYear`), specified as a scalar. This value corresponds to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. The value does not correspond to the radio flux at 1 AU.

Data Types: `double`

F107 — Daily F10.7 flux for previous day
scalar

Daily F10.7 flux for previous day, specified as a scalar. This value corresponds to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. The value does not correspond to the radio flux at 1 AU.

Data Types: `double`

aph — Magnetic index information
1-by-7 array

Magnetic index information, specified as an 1-by-7 array. This information consists of:

- Daily magnetic index (AP)
- 3-hour AP for current time
- 3-hour AP for 3 hours before current time
- 3-hour AP for 6 hours before current time
- 3-hour AP for 9 hours before current time
- Average of eight 3-hour AP indices from 12 to 33 hours before current time
- Average of eight 3-hour AP indices from 36 to 57 hours before current time

  For more information, see “Limitations”.

Data Types: `double`

Parameters

Space weather data file — MAT-file of space weather data

`aeroSpaceWeatherData.mat` (default) | MAT-file name

MAT-file of space weather data. This file is the output from the `aeroReadSpaceWeatherData` function.

Aerospace Blockset includes a default space weather data file, `aeroSpaceWeatherData.mat`. To use the most recent data available, use `aeroReadSpaceWeatherData` to generate a new MAT-file, and specify the file name for this parameter. For more information, see `aeroReadSpaceWeatherData`.

If the file is not on the MATLAB path, specify the full pathname. The MAT-file must contain these variables from the space weather data file:

- `YEAR`
- `MONTH`
- `DAY`
- `AP1`
- `AP2`
- `AP3`
- `AP4`
- `AP5`
- `AP6`
- `AP7`
- `AP8`
- `AP_AVG`
- `F107_OBS`
- `F107_DATA_TYPE`
- `F107_OBS_CENTER81`

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `SpaceWeatherDataFile`
Type: character vector
Values: `'aeroSpaceWeatherData.mat'` | MAT-file name
Default: `'aeroSpaceWeatherData.mat'`

Input time specification — Input time format

`Year, day, sec` (default) | `Julian date`

Input time format, specified as:

- `Year, day, sec` — Enter the year, day, and seconds through the **Year**, **DOY**, and **UT (sec)** input ports.
- `Julian date` — Enter the Julian date through the $\mathbf{T_{JD}}$ input port.

Programmatic Use
Block Parameter: `InputSpec`
Type: character vector
Values: `'Year, day, sec.'` | `'Julian date'`
Default: `'Year, day, sec.'`

F10.7 extrapolation method — Extrapolation method for `f107Average` and `f107Daily`

`None - clip` (default) | `Constant` | `Least squares fit`

Extrapolation method for `f107Average` and `f107Daily` for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`None - clip`	Set `f107Average` and `f107Daily` to the nearest data point available in the MAT-file.
`Constant`	Set `f107Average` and `f107Daily` to a constant value specified by the **Magnetic index extrapolation method** parameter value.
`Least squares fit`	Approximate `f107Average` and `f107Daily` using a least-squares fit of the space weather data from October 1, 1957, to December 1, 2040. This method uses a trigonometric function of the form: `a + b*cos(c*t + d*sin(e*t))`, where: • *a* is `128.2351780622538` • *b* is `54.3285872213434` • *c* is `0.0015460708364` • *d* is `0.2429462096495` • *e* is `0.0015563188188` • *t* is *jdCurrent* - *jdReftime* • *jdCurrent* is the current Julian date • *jdRef* is `2436112.5`

Note For `f107Average` values,if part of the 81-day period falls inside the MAT-file range of space weather data, the Solar Flux and Geomagnetic Index block uses the actual daily values from the overlapping portion. To calculate the average for the nonoverlapping portion, the block uses the clipped, constant, or least squares daily value.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `F107ExtrapMethod`
Type: character vector
Values: `'None - clip'` | `'Constant'` | `'Least squares fit'`
Default: `'None - clip'`

F10.7 extrapolation value — Extrapolation value to assign to `f107Average` and to calculate `f107Daily`

`150.0` (default) | scalar

Extrapolation value to assign to `f107Average` and to calculate `f107Daily`, specified as a scalar.

Tunable: Yes

Dependencies

This value is assigned when:

- **F10.7 extrapolation method** is set to `Constant`.
- Time specified by the **Year**, **DOY**, and **UT (sec)** ports is outside the range of the data in the MAT-file.

Programmatic Use

Block Parameter: `F107ExtrapValue`
Type: character vector
Values: `'150.0'` | scalar
Default: `'150.0'`

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

- Set **Atmospheric model** to `NRLMSISE-00`.
- Set **F10.7 extrapolation method** to `Constant`.

Magnetic index extrapolation method — Extrapolation method for magnetic index information

`None - clip` (default) | `Constant` | `IGRF`

Extrapolation method for magnetic index information values for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`None - clip`	Set elements of magnetic index information to the nearest data point available in the MAT-file.
`Constant`	Set elements of magnetic index information to a constant value specified by the **Magnetic index extrapolation method** parameter. The elements of magnetic index for times outside this range are based on clipped values of the horizontal magnetic field strength at these time limits.
`IGRF`	Calculate the elements of magnetic index information using the International Geomagnetic Reference Field. Because this model is defined for times between January 1, 1900, 12:00 AM UTC and January 1, 2025, 12:00 AM UTC, the predictions for times outside this range are clipped to values at these time limits.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `MagneticIndexExtrapMethod`
Type: character vector
Values: `'None - clip'` | `'Constant'` | `'IGRF'`
Default: `'None - clip'`

Magnetic index extrapolation value — Extrapolation value used to calculate magnetic index elements

`4.0` (default) | scalar

Extrapolation value used to calculate magnetic index elements, specified as a scalar.

Tunable: Yes

Dependencies

This parameter is enabled when **Magnetic index extrapolation method** is set to `Constant`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `MagneticIndexExtrapValue`
Type: character vector
Values: `'4.0'` | scalar
Default: `'4.0'`

Algorithms

The default space weather data file `aeroSpaceWeatherData.mat` installed with the Aerospace Blockset was created using the `aeroReadSpaceWeatherData` function. `aeroReadSpaceWeatherData` reads space weather data from a CelesTrak® consolidated space weather data file into a MAT-file.

- Geomagnetic data - Provided by the Geomagnetic Observatory Niemegk, GFZ German Research Center for Geosciences, and licensed under the CC-BY-4.0 license. Matzka, J., Stolle, C., Yamazaki, Y., Bronkalla, O. and Morschhauser, A., 2021. The geomagnetic Kp index and derived indices of geomagnetic activity. Space Weather, `https://doi.org/10.1029/2020SW002641`.
- Solar Radio Flux Data - Provided by Dominion Radio Astrophysical Observatory and Natural Resources Canada.
- 45-Day and Monthly predictions - Provided by the Space Weather Prediction center of the National Oceanic and Atmospheric Administration (NOAA).

To generate an up-to-date space weather data MAT-file to use with the Solar Flux and Geomagnetic Index block, use the `aeroReadSpaceWeatherData` function.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

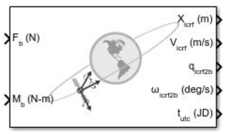
Generate C and C++ code using Simulink® Coder™.

See Also

NRLMSISE-00 Atmosphere Model | `aeroReadSpaceWeatherData`

Spacecraft Dynamics

Model dynamics of one or more spacecraft



Libraries:
Aerospace Blockset / Spacecraft / Spacecraft Dynamics

Description

The Spacecraft Dynamics block models translational and rotational dynamics of spacecraft using numerical integration. It computes the position, velocity, attitude, and angular velocity of one or more spacecraft over time. For the most accurate results, use a variable step solver with low tolerance settings (less than 1e-8). To trade off accuracy for speed, use larger tolerances, depending on your mission requirements.

You can define initial orbital states as

- A set of orbital elements.
- Position and velocity state vectors.

To propagate orbital states, the block uses the gravity model selected for the current central body. It also includes external accelerations and forces that you provide as inputs to the block. To define initial attitude states, use quaternions, direction cosine matrices (DCMs), or Euler angles.

To propagate attitude states, the block uses moments provided as inputs to the block and mass properties defined on the block.

Aerospace Blockset uses quaternions that are defined using the scalar-first convention.

The Spacecraft Dynamics block supports scalar and vector expansion. The block parameter and input port dimensions determine the number of the output signals and the number of spacecraft. After scalar and vector expansion, all parameters in the **Orbit**, **Mass**, and **Attitude** tabs and all input ports except for **φθψ** (Moon libration angles) and **αδW** (right ascension, declination, and rotation angle) input ports are defined for each spacecraft.

The size of the provided initial conditions determines the number of spacecraft being modeled. If you supply more than one value for a parameter in the **Orbit**, **Attitude**, or **Mass** tabs, the block outputs a constellation of satellites. Any parameter with a single provided value is expanded and applied to all the satellites in the constellation. For example, if you provide a single value for all the parameters on the block except **True anomaly**, which contains six values, the block creates a constellation of six satellites, varying true anomaly only.

The block applies the same expansion behavior to the block input ports. All input ports support expansion except **Moon libration angles** (when **Central body** is `Moon`) and **Spin axis right ascension (RA) at J2000**, **Spin axis declination (Dec) at J2000**, and **Initial rotation angle at J2000** (when **Central body** is `Custom`). All other ports accept either a single value expanded to all spacecraft being modeled, or individual values applied to each spacecraft.

For more information on the coordinate systems and rotational and translational dynamics the Spacecraft Dynamics block uses, see “Algorithms” on page 5-1204.

Atmospheric Drag

To help model the drag on spacecraft for high precision orbit propagation, the Spacecraft Dynamics block supports drag. Atmospheric drag affects spacecraft flying at low Earth orbit (LEO); it is less relevant further away from Earth. For the atmospheric drag equation, see “Atmospheric Drag” on page 5-1208.

Third Body Gravity Effects

To include the effects of point-mass “Third Body” on page 5-1191 gravity, select which third bodies to include. These calculations require an **Ephemeris model** and data. For the third body contribution equation, see “Third Body” on page 5-1208.

Solar Radiation Pressure

To include the effects of solar radiation pressure (“SRP” on page 5-1199) on the spacecraft, provide an eclipse fraction or specify a shadow model to compute the fraction. Consider taking solar radiation pressure into account when atmosphere drag is negligible and pointing accuracy requirements are strict. For the solar radiation pressure equation, see “Solar Radiation Pressure” on page 5-1209.

Ports

Input

F_b — Applied forces
3-element vector | *numSat*-by-3 array

Force applied to the spacecraft center of mass in the body frame, specified as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

To enable this port, select the **Input body forces** check box.

Data Types: `double`

M_b — Applied moments
3-element vector | *numSat*-by-3 array

Moment applied to the spacecraft with respect to mass in the body frame, specified as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

To enable this port, select the **Input body moments** check box.

Data Types: `double`

A — External acceleration
3-element vector | *m*-by-3 array

External acceleration to apply to the spacecraft with respect to the ICRF or fixed-frame at the current timestep, specified as a 3-element vector or *m*-by-3 array.

Dependencies

To enable this port, select the **Input external accelerations** check box.

To specify the acceleration coordinate frame, set the **External acceleration coordinate frame** parameter.

Data Types: `double`

φθψ — Moon libration angles
3-element vector

Moon libration angles for transformation between the ICRF and Moon-centric fixed-frame using the Moon-centric Principal Axis (PA) system, specified as a 3-element vector. To get these values, use the Moon Libration block.

Note The fixed-frame used by this block when **Central body** is set to `Moon` is the Mean Earth/pole axis (ME) system. For more information, see "Algorithms" on page 5-669.

Dependencies

To enable this port:

- Set **Central body** to `Moon`.
- Select the **Input Moon libration angles** check box.

Data Types: `double`

αδW — Right ascension, declination, and rotation angle
3-element vector

Central body spin axis instantaneous right ascension, declination, and rotation angle, specified as a 3-element vector. This port is available only for custom central bodies.

Dependencies

To enable this port:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Port`.

Data Types: `double`

m — Spacecraft mass
scalar | 1D array of size *numSat*

Spacecraft mass at the current timestep. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

dm/dt — Rate of change of mass
scalar | 1D array of size *numSat*

Rate of change of mass (positive if accreted, negative if ablated) at the current timestep, specified as a scalar or 1D array of size *numSat*. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Simple Variable`.

Data Types: `double`

I — Spacecraft inertia tensor
3-by-3 array | 3-by-3-by-*numSat* array

Spacecraft inertia tensor, specified as a 3-by-3 array or 3-by-3-by-*numSat* array at the current timestep. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

dI/dt — Rate of change of inertia tensor matrix
3-by-3 array | 3-by-3-by-*numSat* array

Rate of change of inertia tensor matrix, specified as a 3-by-3 array or 3-by-3-by-*numSat* array at the current time step. *numSat* is the number of spacecraft.

Dependencies

To enable this port, set **Mass type** to `Custom Variable`.

Data Types: `double`

V_{re} — Relative velocity
3-element vector | *numSat*-by-3 array

Relative velocity at which the mass is accreted to or ablated from the body in body-fixed axes, specified as a 3-element vector or *numSat*-by-3 array. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set **Mass type** to `Custom Variable` or `Simple Variable`.
- Select the **Include mass flow relative velocity** check box.

Data Types: `double`

ρ — Atmospheric density
scalar

Atmospheric density to calculate acceleration due to atmospheric drag.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Port`.

Data Types: `double`

F107a — 81-day average Ottawa F10.7 cm solar flux
scalar

81-day average Ottawa F10.7 cm solar flux, centered on the current day specified in **Start date/time**. These F107 Average values correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. This site provides both classes of values:

https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Data Types: `double`

F107 — Daily Ottawa F10.7 cm solar flux
scalar

Daily Ottawa F10.7 cm solar flux, centered on the current day specified in **Start date/time**. The f107Daily values do not correspond to the radio flux at 1 AU. This site provides both classes of values:

https://www.ngdc.noaa.gov/stp/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Data Types: `double`

aph — Daily magnetic index information
N-by-7 array

Daily magnetic index information (aph), specified as an *N*-by-7 array. The magnetic index information consists of:
Daily magnetic index (AP)

3 hour AP for current time
3 hour AP for 3 hours before current time
3 hour AP for 6 hours before current time
3 hour AP for 9 hours before current time
Average of eight 3 hour AP indices from 12 to 33 hours before current time
Average of eight 3 hour AP indices from 36 to 57 hours before current time

The effects of daily magnetic index are not large or established below 80,000 m. For more information, see Limitations on NRLMSISE-00 Atmosphere Model.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Data Types: `double`

Flags — Variation flags
array of 23

Variation flags, specified as an array of 23, to enable or disable particular variations for the outputs. You can specify one of the following values for a field. The default value for each field is `1`.

- 0.0

  Removes the value effect on the output.
- 1.0

  Applies the main and the cross-term effects of that value on the output.
- 2.0

  Applies only the cross-term effect of that value on the output.

Field	Description
`Flags(1)`	F10.7 effect on mean
`Flags(2)`	Independent of time
`Flags(3)`	Symmetrical annual
`Flags(4)`	Symmetrical semiannual
`Flags(5)`	Asymmetrical annual
`Flags(6)`	Asymmetrical semiannual
`Flags(7)`	Diurnal
`Flags(8)`	Semidiurnal
`Flags(9)`	Daily AP. If you set this field to -1, the block uses the entire matrix of magnetic index information (APH) instead of `APH(:,1)`
`Flags(10)`	All UT, longitudinal effects

Field	Description
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Flags source** parameter to `Port`.

Data Types: `double`

C_d — Atmospheric drag coefficient
scalar | vector of size *numSat*

Atmospheric drag coefficient, specified as a scalar or vector of size *numSat*.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag coefficient source** parameter to `Port`.

Data Types: `double`

A_d — Atmospheric drag area
scalar | vector of size *numSat*

Atmospheric drag area, specified as a scalar or vector of size *numSat*.

Dependencies

To enable this port:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag area source** parameter to `Port`.

Data Types: `double`

$R_{cb,3}$ — Position of one or more custom bodies
3-element vector | *numCustom3rdBodies*x3-element vector

Position of one or more custom bodies with respect to the current central body, in the ICRF coordinate frame. *numCustom3rdBodies* is the number of custom bodies. For more information, see **Custom gravitational parameter (m^3/s^2)**.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Include third body point-mass gravity** check box.
- Select the **Custom** check box.

Data Types: `double`

Fraction — Fraction of solar disk
between `0` (full eclipse) and `1` (full sunlight) | scalar | *numSat*

Fraction of solar disk visible from the spacecraft position. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.
- Set **Eclipse fraction source** to `Port`.

Data Types: `double`

R_C — Reflectivity coefficient
scalar or *numSat* of values in the range $1 \le R_C \le 2$

Reflectivity coefficient, specified as a scalar or *numSat* of values in the range $1 \le R_C \le 2$. *numSat* is the number of spacecraft.

Dependencies

To enable this port:

- Set **Propagation method** to `Numerical (high precision)`.

- Select the **Include solar radiation pressure (SRP)** check box.
- Set the **Reflectivity coefficient source** parameter to `Port`.

Data Types: `double`

A_{srp} — Cross section area of the spacecraft
scalar | *numSat*

Cross section area of the spacecraft seen by the Sun, specified as a scalar or *numSat*. The block uses this value to calculate the acceleration due to solar radiation pressure.

Dependencies

To enable this port:

- Set **Propagation method** parameter to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.
- Set **SRP area source** parameter to `Port`.

Data Types: `double`

Output

X — Position of spacecraft
3-element vector | *numSat*-by-3 array

Position of the spacecraft with respect to the ICRF or fixed-frame outport coordinate frame, returned as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

- To change the output coordinate frame for this port, set the **State vector output coordinate frame** parameter.
- The size of the initial conditions provided in the **Mass**, **Orbit**, or **Attitude** tab control the port dimension.

Data Types: `double`

V — Velocity
3-element vector | *numSat*-by-3 array

Velocity of the spacecraft with respect to the ICRF or fixed-frame outport coordinate frame, returned as a 3-element vector or *numSat*-by-3 array at the current time step. *numSat* is the number of spacecraft.

Dependencies

- To change the output coordinate frame for this port, set the **State vector output coordinate frame** parameter.
- The size of the initial conditions provided in the **Mass**, **Orbit**, or **Attitude** tab control the port dimension.

Data Types: `double`

A — Total inertial acceleration
3-element vector | *numSat*-by-3 array

Total inertial acceleration of the spacecraft with respect to the ICRF, returned as a 3-element vector or *numSat*-by-3 array at the current timestep. *numSat* is the number of spacecraft.

Dependencies

- To enable this port, select the **Output total inertial acceleration** check box
- The size of the initial conditions provided in the **Orbit** tab control the port dimension.

Data Types: `double`

$\mathbf{q_{body2icrf}}$ — Spacecraft attitude quaternion
4-element quaternion | *numSat*-by-4 array

Spacecraft attitude quaternion, returned as a (scalar first) quaternion rotation from the body axis to the outport frame, as a 4-element quaternion, or *numSat*-by-4 array (scalar first) at the current time step. *numSat* is the number of spacecraft.

Dependencies

The coordinate frame and attitude format of this port depends on these settings:

- To specify the attitude reference coordinate frame, set the **Attitude reference coordinate frame** parameter.
- Set **Attitude representation** to `Quaternion`.

Data Types: `double`

DCM — Spacecraft attitude direction cosine matrix
3-by-3 array | *numSat*-by-3-by-3 array

Spacecraft attitude direction cosine matrix (DCM), returned as a3-by-3 array or *numSat*-by-3-by-3 array. *numSat* is the number of spacecraft.

Dependencies

The coordinate frame and attitude format of this port depends on these settings:

- To specify the attitude reference coordinate frame, set the **Attitude reference coordinate frame** parameter.
- Set **Attitude representation** to `DCM`.

Data Types: `double`

R1,R2,R3 — Spacecraft attitude Euler angles
3-element vector | *numSat*-by-3 array

Spacecraft attitude Euler angles, returned as a 3-element vector or *numSat*-by-3 array. *numSat* is the number of spacecraft.

Dependencies

The coordinate frame and attitude format of this port depend on these settings:

- To specify the attitude reference coordinate frame, set the **Attitude reference coordinate frame** parameter.
- Set **Attitude representation** to `Euler angles`.

Data Types: `double`

ω — Angular rate of spacecraft
3-element vector | *numSat*-by-3 array

Angular rate of the spacecraft relative to the attitude reference coordinate frame, returned as a 3-element vector or *numSat*-by-3 array, expressed as body axis angular rates PQR. *numSat* is the number of spacecraft.

Dependencies

The attitude reference coordinate frame depends on the **Attitude reference coordinate frame** parameter.

Data Types: `double`

dω/dt — Body angular acceleration
3-element array | *numSat*-by-3 array

Body angular acceleration relative to the ICRF frame, returned as a 3-element array or *numSat*-by-3 array. *numSat* is the number of spacecraft.

Dependencies

To enable this port, select the **Output total inertial angular acceleration** check box.

The attitude reference coordinate frame depends on the **Attitude reference coordinate frame** parameter.

Data Types: `double`

$q_{icrf2ff}$ — Coordinate system transformation
4-element array

Coordinate system transformation between the ICRF and fixed-frame coordinate system at the current timestep, returned as a 4-element array.

Dependencies

To enable this port, select the **Output quaternion (ICRF to Fixed-frame)** check box.

Data Types: `double`

t_{utc} — Time at current time step
scalar | 6-element array

Time at current time step, returned as a:

- scalar — If you specify the **Start data/time** parameter as a Julian date.
- 6-element array — If you specify the **Start data/time** parameter as a Gregorian date with six elements (year, month, day, hours, minutes, seconds).

This value equals the **Start date/time** parameter value plus the elapsed simulation time.

Dependencies

To enable this parameter, select the **Output current date/time (UTC Julian date)** check box.

Data Types: `double`

Fuel Status — Fuel status
scalar | *numSat*-element array

Fuel tank status at the current timestep, returned as a scalar or *numSat*-element array, returned as:

- `1` — Tank is full.
- `0` — Tank is not full or empty.
- `-1` — Tank is empty.

numSat is the number of spacecraft.

Dependencies

To enable this port:,

- Set **Mass type** to `Simple Variable`.
- Select the **Output fuel tank status** check box.

Data Types: `double`

Parameters

Main

Input body forces — Option to enable external forces

on (default) | off

To enable external forces to be included in the integration of the spacecraft equations of motion in the body frame, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `forcesin`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Input body moments — Option to enable external moments

on (default) | off

To enable external moments to be included in the integration of the spacecraft equations of motion in the body frame, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `momentsIn`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Input external accelerations — Option to input additional force accelerations

off (default) | on

To enable additional external accelerations to be included in the integration of the spacecraft equations of motion, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `accelIn`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

External acceleration coordinate frame — Frame for acceleration input port

`ICRF` (default) | `Fixed-frame`

Frame for acceleration input port **A**, specified as `ICRF` or `Fixed-frame`.

Dependencies

To enable this parameter, select the **Input external accelerations** check box.

Programmatic Use
Block Parameter: `accelFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'`
Default: `'ICRF'`

State vector output coordinate frame — Position and velocity state output port coordinate frame

`ICRF` (default) | `Fixed-frame`

Position and velocity state output port coordinate frame setup, specified as `ICRF` or `Fixed-frame`.

Programmatic Use
Block Parameter: `outportFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'`
Default: `'ICRF'`

Output total inertial acceleration — Option to enable total acceleration port

off (default) | on

Enable the total acceleration output computed by the block with respect to the ICRF or fixed-frame outport coordinate frame. This acceleration includes all external accelerations, forces, and internal environmental accelerations that act on the spacecraft.

Note Do not use this port as part of a simulation loop (in other words, do not feed this output back into the block).

Tunable: Yes

Dependencies

To change the output coordinate frame for this port, set the **State vector output coordinate frame** parameter.

Programmatic Use
Block Parameter: `AccelOut`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Start date/time (UTC Julian date) — Initial start time for simulation

`juliandate (2020, 1, 1, 12, 0, 0)` (default) | valid scalar Julian date | valid Gregorian date including year, month, day, hours, minutes, seconds as 1D or 6-element array for Gregorian dates

Initial start date and time of simulation, specified as a Julian or Gregorian date. The block defines initial conditions using this value.

Tip To calculate the Julian date, use the `juliandate` function.

Tunable: Yes

Dependencies

The data format for this parameter is controlled by the **Time format** parameter.

Programmatic Use
Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1, 12, 0, 0)'` | valid scalar Julian date | valid Gregorian date including year, month, day, hours, minutes, seconds as 1D or 6-element array
Default: `'juliandate(2020, 1, 1, 12, 0, 0)'`

Output current date/time (UTC Julian date) — Option to add output port t_{utc}

on (default) | off

To output the current date or time, select this check box. Otherwise, clear this check box.

Dependencies

The data format for this parameter is controlled by the **Time format** parameter.

Programmatic Use
Block Parameter: `dateOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Action for out-of-range input — Out-of-range block behavior

`Warning` (default) | `Error` | `None`

Out-of-range block behavior action. Specify one of these options.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Mass

Mass type — Spacecraft mass type

`Fixed` (default) | `Simple Variable` | `Custom Variable`

Spacecraft mass type, specified as:

- `Fixed` — Mass and inertia are constant throughout the simulation.
- `Simple Variable` — Mass and inertia vary linearly as a function of mass rate.
- `Custom Variable` — Instantaneous mass, inertia, and inertia rate are inputs to the block.

Programmatic Use
Block Parameter: `massType`
Type: character vector
Values: `'Fixed'` | `'Simple Variable'` | `'Custom Variable'`
Default: `'Fixed'`

Mass — Initial mass of rigid body spacecraft

4.0 (default) | scalar | vector of size *numSat*

Initial mass of rigid body spacecraft, specified as scalar or vector of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set the **Mass type** parameter to either **Fixed** or **Simple variable**.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'4.0'`

Empty mass — Spacecraft empty mass

`3.5` (default) | scalar | vector of size *numSat*

Spacecraft empty (dry) mass, specified as a scalar or vector of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `emptyMass`
Type: character vector
Values: 1D array of size *numSat* | 1D array of size *numSat*
Default: `'3.5'`

Full mass — Spacecraft full mass

4.0 (default) | scalar | vector of size *numSat*

Spacecraft full (wet) mass, specified as a scalar or vector of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `fullMass`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'4.0'`

Inertia tensor — Inertia tensor matrix

`[0.2273, 0, 0; 0 0.2273 0; 0 0 .0040]` (default) | 3-by-3 array | 3-by-3-by-*numSat* array

Initial inertia tensor matrix of the spacecraft, specified, as a 3-by-3 array for a single spacecraft or a 3-by-3-by-*numSat* array for multiple spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Fixed`.

Programmatic Use
Block Parameter: `inertia`
Type: character vector
Values: `'[0.2273, 0, 0; 0 0.2273 0; 0 0 .0040]'` | 3-by-3 array | 3-by-3-by-*numSat* array
Default: `'[0.2273, 0, 0; 0 0.2273 0; 0 0 .0040]'`

Empty inertia tensor — Empty inertia tensor matrix

`[0.1989, 0, 0; 0 0.1989 0; 0 0 .0035]` (default) | 3-by-3 array | 3-by-3-by-*numSat* array

Empty (dry) inertia tensor matrix, specified as a 3-by-3 array for a single spacecraft or a 3-by-3-by-*numSat* array for multiple spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `emptyInertia`
Type: character vector
Values: 3-by-3 array | 3-by-3-by-*numSat* array
Default: `[0.1989, 0, 0; 0 0.1989 0; 0 0 .0035]`

Full inertia tensor — Full inertia tensor matrix

`[0.2273, 0, 0; 0, 0.2273, 0; 0, 0, .0040]` (default) | 3-by-3 array | 3-by-3-by-*numSat* array

Full (wet) inertia tensor matrix, specified as a 3-by-3 array for a single spacecraft or a 3-by-3-by-*numSat* array for multiple spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `fullInertia`
Type: character vector
Values: 3-by-3 array | 3-by-3-by-*numSat* array
Default: `[0.2273, 0, 0; 0, 0.2273, 0; 0, 0, .0040]`

Include mass flow relative velocity — Option to enable mass flow velocity

off (default) | on

To enable mass flow velocity to the block, select this check box. The mass flow velocity is the relative velocity in the body frame at which the mass is accreted or ablated. To disable mass flow velocity to the block, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple variable` or `Custom variable`.

Programmatic Use
Block Parameter: `useMassFlowRelativeVelocity`
Type: character vector
Values: `'on'` | `'off'`
Default: `'off'`

Limit mass flow when mass is empty or full — Option to limit mass flow

on (default) | off

To limit the mass flow when the spacecraft mass is full or empty, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `limitMassFlow`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Output fuel tank status — Option to enable fuel tank status

on (default) | off

To enable fuel tank status, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Mass type** to `Simple variable`.

Programmatic Use
Block Parameter: `outputFuelStatus`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Orbit

Define the initial states of the spacecraft.

Initial state format — Input method for initial states of orbit

`Orbital elements` (default) | `ICRF state vector` | `Fixed-frame state vector`

Input method for initial states of orbit, specified as `Orbital elements`, `ICRF state vector`, or `Fixed-frame state vector`.

Programmatic Use
Block Parameter `stateFormatNum` when `propagator` is set to `High precision (numerical)`
Type: character vector
Values: `'Orbital elements'` | `'Orbital elements'` | `'ICRF state vector'` | `'Fixed-frame state'` when `propagator` is set to `'High precision (numerical)'`
Default: `'Orbital elements'`

Orbit type — Orbit classification

`Keplerian` (default) | `Elliptical equatorial` | `Circular` | `Circular equatorial`

Orbit classification, specified as:

- `Keplerian` — Model elliptical, parabolic, and hyperbolic orbits using six standard Keplerian orbital elements.
- `Elliptical equatorial` — Fully define an equatorial orbit, where inclination is 0 or 180 degrees and the right ascension of the ascending node is undefined.
- `Circular` — Define a circular orbit, where eccentricity is 0 and the argument of periapsis is undefined. To fully define a circular orbit, select `Circular equatorial`.
- `Circular equatorial` — Fully define a circular orbit, where eccentricity is 0 and the argument of periapsis is undefined.

Dependencies

To enable this parameter, set **Initial state format** to `Orbital elements`.

Programmatic Use
Block Parameter: `orbitType`
Type: character vector
Values: `'Keplerian'` | `'Elliptical equatorial'` | `'Circular inclined'` | `'Circular equatorial'`
Default: `'Keplerian'`

Semi-major axis — Half of major axis of ellipse

6786000 (default) | scalar | 1D array of size *numSat*

Half of ellipse major axis, specified as a 1D array of size *numSat*. *numSat* is the number of spacecraft.

- For parabolic orbits, this block interprets this parameter as the periapsis radius (distance from periapsis to the focus point of orbit).
- For hyperbolic orbits, this block interprets this parameter as the distance from periapsis to the hyperbola center.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial state format** to `Orbital elements`.

Programmatic Use
Block Parameter: `semiMajorAxis`
Type: character vector
Values: scalar | 1D array of size *numSat*
Default: `'6786000'`

Eccentricity — Deviation of orbit

0.01 (default) | scalar | value between `0` and `1`, or greater than `1` for Keplerian orbit type | 1D array of size *numSat*

Deviation of the orbit from a perfect circle, specified as a scalar or 1D array of size *numSat*. *numSat* is the number of spacecraft.

If **Orbit** type is set to `Keplerian`, this value can be:

- `1` for parabolic orbit
- Greater than `1` for hyperbolic orbit

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`.
- Set **Orbit type** to `Keplerian` or `Elliptical equatorial`.

Programmatic Use
Block Parameter: `eccentricity`
Type: character vector
Values: `0.01` | scalar | value between `0` and `1`, or greater than `1` for Keplerian orbit type | 1D array of size *numSat*
Default: `'0.01'`

Inclination — Tilt angle of orbital plane

50 (default) | scalar | 1D array of size *numSat* | degrees between 0 and 180 | radians between 0 and pi

Vertical tilt of the ellipse with respect to the reference plane measured at the ascending node, specified as a scalar or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`
- Set **Orbit type** to `Keplerian` or `Circular inclined`

Programmatic Use
Block Parameter: `inclination`
Type: character vector
Values: 50 | scalar | 1D array of size *numSat* | degrees between 0 and 180 | radians between 0 and pi
Default: `'50'`

RAAN — Angular distance in equatorial plane

95 (default) | scalar value between `0` and `360` | 1D array of size *numSat*

Right ascension of ascending node (RAAN), specified as a value between `0` and `360`, specified as a scalar or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft. RAAN is the angular distance along the reference plane from the International Celestial Reference Frame (ICRF) *x*-axis to the location of the ascending node — the point at which the spacecraft crosses the reference plane from south to north.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`.
- Set **Orbit type** to `Keplerian` or `Circular inclined`.

Programmatic Use
Block Parameter: `raan`
Type: character vector
Values: `'95'` | scalar value between `0` and `360` | 1D array of size *numSat*
Default: `'95'`

Argument of periapsis — Angle from spacecraft ascending node to periapsis

93 (default) | value between `0` and `360` | 1D array of size *numSat*

Angle from the spacecraft ascending node to periapsis (closest point of orbit to the central body), specified as a 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`
- Set **Orbit type** to `Keplerian`

Programmatic Use
Block Parameter: `argPeriapsis`
Type: character vector
Values: 93 | scalar value between `0` and `360` | 1D array of size *numSat*
Default: `'93'`

True anomaly — Angle between periapsis and initial position of spacecraft

203 (default) | scalar value between `0` and `360` | 1D array of size *numSat*

Angle between periapsis (closest point of orbit to the central body) and the initial position of spacecraft along its orbit at **Start date/time**, specified as a scalar or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`.
- Set **Orbit type** to `Keplerian` or `Elliptical inclined`.

Programmatic Use
Block Parameter: `trueAnomaly`
Type: character vector
Values: `'203'` | scalar value between `0` and `360` | 1D array of size *numSat*
Default: `'203'`

Argument of latitude — Angle between ascending node and initial position of spacecraft

200 (default) | scalar | value between `0` and `360` | 1D array of size *numSat*

Angle between the ascending node and the initial position of spacecraft along its orbit at **Start date/time**, specified as a scalar or 3-element vector or 1D array of size *numSat*, in specified units. *numSat* is number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`.
- Set **Orbit Type** to `Circular inclined`.

Programmatic Use

Block Parameter: `argLat`
Type: character vector
Values: `'200'` | scalar value between `0` and `360` | 1D array of size *numSat*
Default: `'200'`

Longitude of periapsis — Angle between ICRF *x*-axis and eccentricity vector

100 (default) | scalar | value between `0` and `360` | 1D array of size *numSat*

Angle between the ICRF *x*-axis and the eccentricity vector, specified as a scalar or 3-element vector or 1D array of size *numSat*, in specified units. *numSat* is the number of spacecraft

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`.
- Set **Orbit type** to `Elliptical equatorial`.

Programmatic Use

Block Parameter: `lonPeriapsis`
Type: character vector
Values: 100 | scalar value between `0` and `360` | 1D array of size *numSat*
Default: `'100'`

True longitude — Angle between ICRF *x*-axis and initial position of spacecraft

150 (default) | scalar | value between `0` and `360` | 1D array of size *numSat* | *numSat*-by-3 vector

Angle between the ICRF *x*-axis and the initial position of spacecraft along its orbit at **Start date/time**, specified as a scalar or 1D array of size *numSat* or a *numSat*-by-3 vector, in specified units. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Initial state format** to `Orbital elements`.
- Set **Orbit type** to `Circular equatorial`.

Programmatic Use

Block Parameter: `trueLon`
Type: character vector

Values: `'150'` | scalar value between `0` and `360` | 1D array of size *numSat* | *numSat*-by-3 vector
Default: `'150'`

ICRF position — Cartesian position vector of spacecraft

`[3649700.0 3308200.0 -4676600.0]` (default) | 3-element vector | | *numSat*-by-3 array

Cartesian position vector of spacecraft in ICRF coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or a *numSat*-by-3 array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial state format** to `ICRF state vector`.

Programmatic Use

Block Parameter: `inertialPosition`
Type: character vector
Values: `[3649700.0 3308200.0 -4676600.0]` | 3-element vector | *numSat*-by-3 array
Default: `'[3649700.0 3308200.0 -4676600.0]'`

ICRF velocity — Cartesian velocity vector of spacecraft

`[-2750.8 6666.4 2573.4]` (default) | 3-element vector | *numSat*-by-3 array

Cartesian velocity vector of spacecraft in ICRF coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or a *numSat*-by-3 array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial state format** to `ICRF state vector`.

Programmatic Use

Block Parameter: `inertialVelocity`
Type: character vector
Values: `[-2750.8 6666.4 2573.4]` | 3-element vector | 2-D array of size *numSat*-by-3 array
Default: `'[-2750.8 6666.4 2573.4]'`

Fixed-frame position — Position vector of spacecraft

[-4142689.0 -2676864.7 -4669861.6] (default) | 3-element vector | *numSat*-by-3 array

Cartesian position vector of spacecraft in fixed-frame coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or a *numSat*-by-3 array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial state format** to `Fixed-frame state vector`.

Programmatic Use
Block Parameter: `fixedPosition`
Type: character vector
Values: `'[-4142689.0 -2676864.7 -4669861.6]'` | 3-element vector for single spacecraft | *numSat*-by-3 array
Default: `'[-2750.8 6666.4 2573.4]'`

Fixed-frame velocity — Velocity vector of spacecraft

[1452.7 -6720.7 2568.1] (default) | 3-element vector | *numSat*-by-3 array

Cartesian velocity vector of spacecraft in fixed-frame coordinate system at **Start date/time**, specified as a 3-element vector for single spacecraft or a *numSat*-by-3 array for multiple spacecraft. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial state format** to `Fixed-frame state vector`.

Programmatic Use
Block Parameter: `fixedVelocity`
Type: character vector
Values: `'[1452.7 -6720.7 2568.1]'` | 3-element vector | *numSat*-by-3 array
Default: `'[1452.7 -6720.7 2568.1]'`

Attitude

Attitude reference coordinate frame — Attitude and angular rate coordinate frame

`ICRF` (default) | `Fixed-frame` | `NED` | `LVLH`

Attitude and angular rate coordinate frame with respect to the attitude and angular rate initial conditions, specified as:

- `ICRF`
- `Fixed-frame`
- `NED`
- `LVLH`

Programmatic Use
Block Parameter: `attitudeFrame`
Type: character vector
Values: `'ICRF'` | `'Fixed-frame'` | `'NED'` | `'LVLH'`
Default: `'ICRF'`

Attitude representation — Orientation format

`Quaternion` (default) | `DCM` | `Euler angles`

Orientation format for spacecraft attitude (initial condition and output port), specified as `Quaternion`, `DCM`, or `Euler angles`.

Programmatic Use

Block Parameter: `attitudeFrame`
Type: character vector
Values: `'Quaternion'` | `'DCM'` | `'Euler angles'`
Default: `'Quaternion'`

Initial body attitude — Spacecraft initial attitude

`[1, 0, 0, 0]` (default) | 4-element vector | *numSat*-by-4 array | 3-by-3 array | *numSat*-by-3-by-3 array

Spacecraft initial attitude (orientation) of the spacecraft provided as either a quaternion, DCM, or Euler angle set with respect to **Attitude representation**.

Tunable: Yes

Dependencies

This parameter name and value format changes depending on the **Attitude representation** parameter.

Parameter Name	Attitude Representation Setting	Value Format
Initial quaternion	`Quaternion`	• 4-element vector • *numSat*-by-4 array
Initial DCM	`DCM`	• 3-by-3 array • *numSat*-by-3-by-3 array
Initial Euler angles	`Euler angles`	• 3-element vector • *numSat*-by-3 array

Programmatic Use

Block Parameter: `attitude`
Type: character vector
Values: 4-element vector | *numSat*-by-4 array | 3-by-3 array | *numSat*-by-3-by-3 array | 3-element array | *numSat*-by-3 array
Default: `'[1, 0, 0, 0]'`

Angle rotation order — Angle rotation order

ZYX (default) | ZYX | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation angle sequence for Euler angle attitude representation.

Tunable: Yes

Dependencies

To enable this parameter, set **Attitude representation** to `Euler angles`.

Programmatic Use

Block Parameter: `rotationOrder`
Type: character vector

Values: 'ZYX' | 'ZYZ' |'ZXY' | 'ZXZ' | 'YXZ' | 'YXY' | 'YZX' | 'YZY' | 'XYZ' | 'XYX' | 'XZY' | 'XZX'
Default: 'ZYX'

Initial body angular rates PQR — Initial body-fixed angular rates

[0, 0, 0] (default) | 3-element vector | *numSat*-by-3 array

Initial body-fixed angular rates (PQR) with respect to **Attitude reference coordinate frame**.

Tunable: Yes

Programmatic Use
Block Parameter: attitudeRate
Type: character vector
Values: | 3-element vector | *numSat*-by-3 array
Default: [0, 0, 0]

Output total inertial angular acceleration — Option to enable total vehicle acceleration

off (default) | on

Enable output total vehicle acceleration computed by the block with respect to the ICRF attitude reference coordinate frame. This acceleration includes all moments that act on the spacecraft.

Tunable: Yes

Programmatic Use
Block Parameter: angAccelOut
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Include gravity gradient torque — Option to enable gravity gradient torque

on (default) | off

Select this check box to enable the use of the gravity gradient torque in the block rotational dynamics equations. Otherwise, clear this check box.

Tunable: Yes

Programmatic Use
Block Parameter: angAccelOut
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Central Body

Central body — Celestial body around which spacecraft orbits

Earth (default) | Moon | Mercury | Venus | Mars | Jupiter | Saturn | Uranus | Neptune | Sun | Custom

Celestial body, specified as Earth, Moon, Mercury, Venus, Mars, Jupiter, Saturn, Uranus, Neptune, Sun, or Custom, around which the spacecraft defined in the **Orbit** tab orbits.

Programmatic Use

Block Parameter: centralBody
Type: character vector
Values: 'Earth' | 'Moon' |'Mercury' | 'Venus' | 'Mars' | 'Jupiter' | 'Saturn' | 'Uranus' | 'Neptune' | 'Sun' | 'Custom' |
Default: 'Earth'

Gravitational potential model — Gravity model for central body

Spherical harmonics when **Central body** set to Earth, Moon, Mars, or Custom, Oblate ellipsoid when **Central body** set to Mercury, Venus, Jupiter, Saturn, Uranus, or Neptune (default) | Point-mass | Oblate ellipsoid (J2)

Control the gravity model for the central body by specifying as Spherical harmonics, Point-mass, or Oblate ellipsoid (J2).

Dependencies

Available options are based on **Central body** settings.

Earth, Moon, Mars, or Custom	Mercury, Venus, Jupiter, Saturn, Uranus, or Neptune
Spherical harmonics	Oblate ellipsoid (J2)
Point-mass	Point-mass
Oblate ellipsoid (J2)	—

Programmatic Use

Block Parameter: gravityModel when centralBody set to 'Earth', 'Moon', 'Mars', or 'Custom' | gravityModelnoSH when centralBody set to Mercury, Venus, Jupiter, Saturn, Uranus, or Neptune
Type: character vector
Values: 'Spherical harmonics' | 'Point-mass' | 'Oblate ellipsoid (J2)' when centralBody set to 'Earth', 'Moon', 'Mars', or 'Custom'; 'Point-mass' | 'Oblate ellipsoid (J2)' when centralBody set to Mercury, Venus, Jupiter, Saturn, Uranus, or Neptune
Default: 'Spherical harmonics' when centralBody set to 'Earth', 'Moon', 'Mars', or 'Custom'; 'Oblate ellipsoid (J2)' when centralBody set to Mercury, Venus, Jupiter, Saturn, Uranus, or Neptune

Spherical harmonic model — Spherical harmonic model

EGM2008 for **Central body** set to Earth, LP-100K for **Central body** set to Moon, GMM2B for **Central body** set to Mars, (default) | EGM96 | EIGEN-GL04C | LP-165P

Spherical harmonic gravitational potential model, specified according to the specified **Central body**.

Dependencies

Available options are based on **Central body** settings:

Central body	Spherical Harmonic Model Option
Earth	EGM2008, EGM96, or EIGEN-GL04C
Moon	LP-100K or LP-165P
Mars	GMM2B

Programmatic Use
Block Parameter: `'earthSH'` when `centralBody` set to `'Earth'` | `'moonSH'` when `centralBody` set to `'Moon'` | `'marsSH'` when `centralBody` set to `'Mars'`
Type: character vector
Values: `'EGM2008'` | `'EGM96'` | `'EIGEN-GL04C'` when `centralBody` set to `'earthSH'`; `'LP-100K'` | `'LP-165P'` when `centralBody` set to `'moonSH'`; `'GMM2B'` when `centralBody` set to `'marsSH'`
Default: `'Spherical harmonics'`

Rotational rate — Rotational rate

4.06124975e-3 (default) | scalar

Rotational rate of a custom central body, specified as a scalar.

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `'customOmega'`
Type: character vector
Values: `'4.06124975e-3'` | scalar
Default: `'4.06124975e-3'`

Spherical harmonic coefficient file — Harmonic coefficient MAT-file

`aerogmm2b.mat` (default) | harmonic coefficient MAT-file

Harmonic coefficient MAT-file that contains definitions for a custom planetary model, specified as a character vector or string.

This file must contain these variables:

Variable	Description
Re	Scalar of planet equatorial radius in meters (m).
GM	Scalar of planetary gravitational parameter in meters cubed per second squared (m^3/s^2).
degree	Scalar of maximum degree.
C	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *C*.
S	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *S*.

Dependencies

To enable this parameter:

- Set**Central body** to `Custom`.
- Set **Gravitational potential model** to `Spherical harmonics`.

Programmatic Use
Block Parameter: `shFile`
Type: character vector
Values: `'aerogmm2b.mat'` | harmonic coefficient MAT-file
Default: `'aerogmm2b.mat'`

Degree — Degree of harmonic model

`120` (default) | scalar | maximum of 2159

Degree of harmonic model, specified as a scalar.

Planet Model	Recommended Degree	Maximum Degree
`EGM2008`	120	2159
`EGM96`	70	360
`LP100K`	60	100
`LP165P`	60	165
`GMM2B`	60	80
`EIGENGL04C`	70	360

Dependencies

To enable this parameter:

- Set **Central body** to `Earth`, `Moon`, `Mars`, or `Custom`.
- Set **Gravitational potential model** to `Spherical harmonics`.

Programmatic Use
Block Parameter: `shDegree`
Type: character vector
Values: `'80'` | scalar
Default: `'80'`

Use Earth orientation parameters (EOPs) — Option to use Earth orientation parameters

on (default) | off

Select this check box to use Earth orientation parameters for the transformation between the ICRF and fixed-frame coordinate systems. Otherwise, clear this check box.

Dependencies

To enable this parameter, set **Central body** to `Earth`.

Additionally, it must satisfy one of these criteria:

- **Gravitational potential model** is set to either `Spherical harmonics` or `Oblate ellipsoid (J2)`.

- **External acceleration coordinate frame** is set to `Fixed-frame`.
- **State vector output coordinate frame** is set to `Fixed-frame`.
- **Attitude reference coordinate frame** is set to `Fixed-frame` or `NED`.

Programmatic Use
Block Parameter: `useEOPs`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

IERS EOP data file — Earth orientation data

`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

Dependencies

To enable this parameter:

- Select the **Use Earth orientation parameters (EOPs)** check box.
- Set **Central body** to `Earth`.

Programmatic Use
Block Parameter: `eopFile`
Type: character vector
Values: `'aeroiersdata.mat'` | `MAT-file`
Default: `'aeroiersdata.mat'`

Input Moon libration angles — Moon libration Euler angle rate

off (default) | on

To specify Euler libration angles (φ θ ψ) for Moon orientation, select this check box.

Dependencies

To enable this parameter, set **Central body** to `Moon`.

Programmatic Use
Block Parameter: `useMoonLib`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Output quaternion (ICRF to Fixed-frame) — Option to add output transformation quaternion port

off (default) | on

To add output transformation quaternion port for the quaternion transformation from the ICRF to the fixed-frame coordinate system, select this check box. Otherwise, clear this check box.

Programmatic Use
Block Parameter: `outputTransform`
Type: character vector

Values: `'off'` | `'on'`
Default: `'off'`

Central body spin axis source — Central body spin source

`Port` (default) | `Dialog`

Central body spin axis source, specified as `Port` or `Dialog`. The block uses the spin axis to calculate the transformation from the ICRF to the fixed-frame coordinate system for the custom central body.

Dependencies

To enable this parameter, set **Central body** to `Custom`.

Programmatic Use
Block Parameter: `cbPoleSrc`
Type: character vector
Values: `'Port'` | `'Dialog'`
Default: `'Port'`

Spin axis right ascension (RA) at J2000 — Right ascension of central body spin axis at J2000

`317.68143` (default) | double scalar

Right ascension of central body spin axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRA`
Type: character vector
Values: `'317.68143'` | double scalar
Default: `'317.68143'`

Spin axis RA rate (deg/century) — Right ascension rate of central body spin axis

`-0.1061` (default) | double scalar

Right ascension rate of the central body spin axis, specified as a double scalar, in specified angle units/century.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.

- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRARate`
Type: character vector
Values: `'-0.1061'` | double scalar
Default: `'-0.1061'`

Spin axis declination (Dec) at J2000 — Declination of central body spin axis at J2000

`52.88650` (default) | double scalar

Declination of the central body spin axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbDec`
Type: character vector
Values: `'52.88650'` | double scalar
Default: `'52.88650'`

Spin axis Dec rate (deg/century) — Declination rate of central body spin axis

`-0.0609` (default) | double scalar

Declination rate of the central body spin axis, specified as a double scalar, in specified angle units/century.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbDecRate`
Type: character vector
Values: `'-0.0609'` | double scalar
Default: `'-0.0609'`

Initial rotation angle at J2000 — Rotation angle of central body *x*-axis

`176.630` (default) | double scalar

Rotation angle of the central body x axis with respect to the ICRF *x*-axis at J2000 (2451545.0 JD, 2000 Jan 1 12:00:00 TT), specified as a double scalar, in specified angle units.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRotAngle`
Type: character vector
Values: `'176.630'` | double scalar
Default: `'176.630'`

Rotation rate (deg/day) — Rotation rate of central body *x*-axis

`350.89198226` (default) | double scalar

Rotation rate of the central body x axis with respect to the ICRF *x*-axis (2451545.0 JD, 2000 Jan 1 12:00:00 UTC), specified as a double scalar, in angle units/day.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Central body spin axis source** to `Dialog`.

Programmatic Use
Block Parameter: `cbRotRate`
Type: character vector
Values: `'350.89198226'` | double scalar
Default: `'350.89198226'`

Equatorial radius — Equatorial radius

`3396200` (default) | double scalar

Equatorial radius for a custom central body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Gravitational potential model** to `None`, `Point-mass`, or `Oblate ellipsoid (J2)`.

Programmatic Use
Block Parameter: `customR`
Type: character vector

Values: `'3396200'` | double scalar
Default: `'3396200'`

Flattening — Flattening ratio

`0.00589` (default) | double scalar

Flattening ratio for custom central body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Gravitational potential model** to `Point-mass`, `Oblate ellipsoid (J2)`, or `Spherical harmonics`.

Programmatic Use
Block Parameter: `customF`
Type: character vector
Values: `'0.00589'` | double scalar
Default: `'0.00589'`

Gravitational parameter — Gravitational parameter

`4.305e13` (default) | double scalar

Gravitational parameter for a custom central body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Gravitational potential model** to `None`, `Point-mass`, or `Oblate ellipsoid (J2)`.

Programmatic Use
Block Parameter: `customMu`
Type: character vector
Values: `'4.305e13'` | double scalar
Default: `'4.305e13'`

Second degree zonal harmonic (J2) — Most significant or largest spherical harmonic term

`1.0826269e-03` (default) | double scalar

Most significant or largest spherical harmonic term, which accounts for oblateness of a celestial body, specified as a double scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set **Central body** to `Custom`.
- Set **Gravitational potential model** to `Oblate ellipsoid (J2)`.

Programmatic Use
Block Parameter: `customJ2`
Type: character vector
Values: `'1.0826269e-03'` | double scalar
Default: `'1.0826269e-03'`

Drag

Configure the atmospheric drag environment.

Include atmospheric drag — Option to include atmospheric drag

off (default) | on

To include atmospheric drag, select this check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.

Programmatic Use
Block Parameter: `useDrag`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Atmospheric density source — Source of atmospheric density value

`Dialog` (default) | `Port`

Source of atmospheric density value, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `atmosSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Atmospheric model — Atmospheric model

`NRLMSISE-00` (default)

Atmospheric model for atmospheric drag calculation, specified as `NRLMSISE-00`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `atmosModel`
Type: character vector
Values: `'NRLMSISE-00'`
Default: `'NRLMSISE-00'`

Flux source — Source of space weather data

`Dialog` (default) | `Port`

Source for historical and predicted flux and geomagnetic indices used by atmospheric density calculation, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `fluxSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Space weather data file — MAT-file of space weather data

`aeroSpaceWeatherData.mat` (default) | MAT-file name

MAT-file of space weather data. This file is the output from the `aeroReadSpaceWeatherData` function.

Aerospace Blockset includes a default space weather data file, `aeroSpaceWeatherData.mat`. To use the most recent data available, use `aeroReadSpaceWeatherData` to generate a new MAT-file, and specify the file name for this parameter. For more information, see `aeroReadSpaceWeatherData`.

If the file is not on the MATLAB path, specify the full pathname. The MAT-file must contain these variables from the space weather data file:

- `YEAR`
- `MONTH`
- `DAY`
- `AP1`
- `AP2`
- `AP3`
- `AP4`
- `AP5`
- `AP6`
- `AP7`
- `AP8`
- `AP_AVG`
- `F107_OBS`
- `F107_DATA_TYPE`
- `F107_OBS_CENTER81`

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `SpaceWeatherDataFile`
Type: character vector
Values: `'aeroSpaceWeatherData.mat'` | MAT-file name
Default: `'aeroSpaceWeatherData.mat'`

F10.7 extrapolation method — Extrapolation method for `f107Average` and `f107Daily`

`None - clip` (default) | `Constant` | `Least squares fit`

Extrapolation method for `f107Average` and `f107Daily` for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`None - clip`	Set `f107Average` and `f107Daily` to the nearest data point available in the MAT-file.

Method	Description
`Constant`	Set `f107Average` and `f107Daily` to a constant value specified by the **Magnetic index extrapolation method** parameter value.
`Least squares fit`	Approximate `f107Average` and `f107Daily` using a least-squares fit of the space weather data from October 1, 1957, to December 1, 2040. This method uses a trigonometric function of the form: `a + b*cos(c*t + d*sin(e*t))`, where: • *a* is `128.2351780622538` • *b* is `54.3285872213434` • *c* is `0.0015460708364` • *d* is `0.2429462096495` • *e* is `0.0015563188188` • *t* is *jdCurrent - jdReftime* • *jdCurrent* is the current Julian date • *jdRef* is `2436112.5`

Note For `f107Average` values,if part of the 81-day period falls inside the MAT-file range of space weather data, the Solar Flux and Geomagnetic Index block uses the actual daily values from the overlapping portion. To calculate the average for the nonoverlapping portion, the block uses the clipped, constant, or least squares daily value.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `F107ExtrapMethod`
Type: character vector
Values: `'None - clip'` | `'Constant'` | `'Least squares fit'`
Default: `'None - clip'`

F10.7 extrapolation value — Extrapolation value to assign to `f107Average` and to calculate `f107Daily`

`150.0` (default) | scalar

Extrapolation value to assign to `f107Average` and to calculate `f107Daily`, specified as a scalar.

Tunable: Yes

Dependencies

This value is assigned when:

- **F10.7 extrapolation method** is set to `Constant`.
- Time specified by the **Year**, **DOY**, and **UT (sec)** ports is outside the range of the data in the MAT-file.

Programmatic Use

Block Parameter: `F107ExtrapValue`
Type: character vector
Values: `'150.0'` | scalar
Default: `'150.0'`

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.
- Set **F10.7 extrapolation method** to `Constant`.

Magnetic index extrapolation method — Extrapolation method for magnetic index information

`None - clip` (default) | `Constant` | `IGRF`

Extrapolation method for magnetic index information values for times outside the range of the MAT-file data, specified as one of these values.

Method	Description
`None - clip`	Set elements of magnetic index information to the nearest data point available in the MAT-file.
`Constant`	Set elements of magnetic index information to a constant value specified by the **Magnetic index extrapolation method** parameter. The elements of magnetic index for times outside this range are based on clipped values of the horizontal magnetic field strength at these time limits.

Method	Description
IGRF	Calculate the elements of magnetic index information using the International Geomagnetic Reference Field. Because this model is defined for times between January 1, 1900, 12:00 AM UTC and January 1, 2025, 12:00 AM UTC, the predictions for times outside this range are clipped to values at these time limits.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `MagneticIndexExtrapMethod`
Type: character vector
Values: `'None - clip'` | `'Constant'` | `'IGRF'`
Default: `'None - clip'`

Magnetic index extrapolation value — Extrapolation value used to calculate magnetic index elements

`4.0` (default) | scalar

Extrapolation value used to calculate magnetic index elements, specified as a scalar.

Tunable: Yes

Dependencies

This parameter is enabled when **Magnetic index extrapolation method** is set to `Constant`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set **Atmospheric model** to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `MagneticIndexExtrapValue`
Type: character vector
Values: `'4.0'` | scalar

Default: '4.0'

Flags source — Variation flag source

Dialog (default) | Port

Variation flag source, specified as Dialog or Port.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to Numerical (high precision).
- Set the **Central Body** parameter to Earth.
- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to Dialog.
- Set the **Atmospheric model** parameter to NRLMSISE-00.

Programmatic Use
Block Parameter: fluxFlagsSrc
Type: character vector
Values: 'Dialog' | 'Port'
Default: 'Dialog'

Flags — Variation flags

ones(1,23) (default)

Variation flags, specified as an array of 23 (ones(1,23)). You can specify one of the following values for a field. The default value for each field is 1.

- 0.0 — Removes the effect on the output.
- 1.0 — Applies the main and the cross-term effects of that value on the output.
- 2.0 — Applies only the cross-term effect of that value on the output.

The array has these fields.

Field	Description
Flags(1)	F10.7 effect on mean
Flags(2)	Independent of time
Flags(3)	Symmetrical annual
Flags(4)	Symmetrical semiannual
Flags(5)	Asymmetrical annual
Flags(6)	Asymmetrical semiannual
Flags(7)	Diurnal
Flags(8)	Semidiurnal
Flags(9)	Daily AP. If you set this field to -1, the block uses the entire matrix of magnetic index information (APH) instead of APH(:,1).

Field	Description
`Flags(10)`	All UT, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Flags source** to `Dialog`.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Atmospheric model** parameter to `NRLMSISE-00`.

Programmatic Use

Block Parameter: `fluxFlags`
Type: character vector
Values: `'ones(1,23)'`
Default: `'ones(1,23)'`

Include anomalous oxygen in density calculation — Option to include anomalous oxygen in density calculation

off (default) | on

To include anomalous oxygen in density calculations, select this check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.

- Select the **Include atmospheric drag** check box.
- Set the **Atmospheric density source** parameter to `Dialog`.
- Set the **Atmospheric model** parameter to `NRLMSISE-00`.

Programmatic Use
Block Parameter: `useOxygen`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Drag coefficient source — Source of drag coefficient

`Dialog` (default) | `Port`

Source of drag coefficient, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `dragCoeffSrc`
Type: character vector
Values: `'Dialog'` | `'Source'`
Default: `'Dialog'`

Drag coefficient — Spacecraft coefficient of drag

2.179 (default) | scalar | vector of size *numSat*

Spacecraft coefficient of drag used by atmospheric drag calculation, specified as a scalar or as a vector of size *numSat*.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag coefficient source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `dragCoeff`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'2.179'`

Drag area source — Source of drag area

`Dialog` (default) | `Port`

Source of drag area, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.

Programmatic Use
Block Parameter: `dragAreaSrc`
Type: character vector
Values: `'Dialog'` | `'Source'`
Default: `'Dialog'`

Drag area — Area to compute acceleration due to atmospheric drag

1.0 (default) | scalar | vector of size *numSat*

Area to compute acceleration due to atmospheric drag, specified as a scalar or as a vector of size *numSat*. This area of the spacecraft is perpendicular to the spacecraft relative velocity.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to `Earth`.
- Select the **Include atmospheric drag** check box.
- Set the **Drag area source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `dragArea`
Type: character vector
Values: scalar | vector of size *numSat*
Default: `'1.0'`

Third Body

Configure third body point mass for gravitational acceleration.

Ephemeris model — Ephemeris model

`DE405` (default) | `DE421` | `DE423` | `DE430` | `DE432t`

Select one of these ephemerides models defined by the Jet Propulsion Laboratory. The block uses ephemeris data to calculate relative celestial positions required for third body point mass gravity and solar radiation pressure.

Ephemeris Model	Description
DE405	Released in 1998. This ephemeris takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE421	Released in 2008. This ephemeris takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050). This block implements these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
DE423	Released in 2010. This ephemeris takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE430	Released in 2013. This ephemeris takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
DE432t	Released in April 2014. This ephemeris takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650). This block implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Note This block requires that you download ephemeris data using the Add-On Explorer. To start the Add-On Explorer, in the MATLAB Command Window, type `aeroDataPackage`. in the MATLAB desktop toolstrip, click **Add-Ons** .

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use

Block Parameter: `ephemerisModel`
Type: character vector
Values: `DE405` | `DE421` | `DE423` | `DE430`
Default: `'DE405'`

Limit ephemerides date range — Option to enable start and end of range of ephemeris data

`off` (default) | `on`

Control how much data is loaded into memory during simulation and how much data is included in generated code for the block:

- Select this check box to limit the loading of ephemeris data to a specified date range.
- Clear this check box to include data for the complete date range defined in the **"Ephemeris model" on page 5-0** table.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `useDateRange`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Start date (JD) — Start date of ephemerides date range

`juliandate(2020, 1, 1)` (default) | Julian date

Start date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use
Block Parameter: `startDate`
Type: character vector
Values: `'juliandate(2020, 1, 1)'` | Julian date
Default: `'juliandate(2020, 1, 1)'`

End date (JD) — End date of ephemerides date range

`juliandate(2050, 1, 1)` (default) | Julian date

End date of ephemerides date range, specified as a Julian date.

Dependencies

To enable this parameter, select the **Limit ephemerides date range** check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Limit ephemerides date range** parameter to `on`.

Programmatic Use
Block Parameter: `endDate`
Type: character vector
Values: `'juliandate(2050, 1, 1)'` | Julian format date
Default: `'juliandate(2050, 1, 1)'`

Include third body point-mass gravity — Acceleration due to third body gravity

`off` (default) | `on`

To include acceleration due to third body gravity in orbit propagation calculation, select this check box. Otherwise, clear this check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use
Block Parameter: `useThirdBodyGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Sun — Gravitational acceleration due to Sun

`on` (default) | `off`

To include gravitational acceleration due to the Sun (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Sun`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeSunGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Mercury — Gravitational acceleration due to Mercury

`off` (default) | `on`

To include gravitational acceleration due to the Mercury (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Mercury`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeMercuryGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Venus — Gravitational acceleration due to Venus

`off` (default) | `on`

To include gravitational acceleration due to the Venus (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Venus`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeVenusGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Earth — Gravitational acceleration due to Venus

`on` (default) | `off`

To include gravitational acceleration due to the Earth (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Earth`.
- Set the **Include third body point-mass gravity** to `on`.

Programmatic Use
Block Parameter: `includeVenusGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Moon — Gravitational acceleration due to Moon

on (default) | off

To include gravitational acceleration due to the Moon (point mass), select this check box. Otherwise, clear it.

Programmatic Use
Block Parameter: includeMoonGravity
Type: character vector
Values: 'off' | 'on' |
Default: 'on'

Mars — Gravitational acceleration due to Mars

off (default) | on

To include gravitational acceleration due to the Mars (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to Numerical (high precision).
- Set the **Central Body** parameter to any value other than Mars.
- Set the **Include third body point-mass gravity** parameter to on.

Programmatic Use
Block Parameter: includeMarsGravity
Type: character vector
Values: 'off' | 'on' |
Default: 'off'

Jupiter — Gravitational acceleration due to Jupiter

off (default) | on

To include gravitational acceleration due to the Jupiter (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to Numerical (high precision).
- Set the **Central Body** parameter to any value other than Jupiter.
- Set the **Include third body point-mass gravity** parameter to on.

Programmatic Use
Block Parameter: includeJupiterGravity
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Saturn — Gravitational acceleration due to Saturn

off (default) | on

To include gravitational acceleration due to the Saturn (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Saturn`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includeSaturnGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Uranus — Gravitational acceleration due to Uranus

off (default) | on

To include gravitational acceleration due to the Uranus (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Uranus`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use

Block Parameter: `includeUranusGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Neptune — Gravitational acceleration due to Neptune

off (default) | on

To include gravitational acceleration due to the Neptune (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Neptune`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeNeptuneGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Pluto — Gravitational acceleration due to Pluto

`off` (default) | `on`

To include gravitational acceleration due to the Pluto (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Central Body** parameter to any value other than `Pluto`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includePlutoGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Custom — Gravitational acceleration due to custom planet

`off` (default) | `on`

To include gravitational acceleration due to a custom planet (point mass), select this check box. Otherwise, clear it.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include third body point-mass gravity** parameter to `on`.

Programmatic Use
Block Parameter: `includeCustomGravity`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Custom gravitational parameter (m^3/s^2) — Custom gravitational acceleration

`42.828314258067e12` (default) | scalar | vector of length *numCustom3rdBodies*

Custom gravitational acceleration for custom third body, specified as a scalar or as a vector of length *numCustom3rdBodies*. *numCustom3rdBodies* is the number of rows provided to the **R_cb** input port. Provide more than one value to include multiple custom bodies.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include third body point-mass gravity** parameter to `on`.
- Set the **Custom** parameter to `on`.

Programmatic Use

Block Parameter: `customThirdBodyMu`
Type: character vector
Values: `'42.828314258067e12'` | scalar | vector of length *numCustom3rdBodies*
Default: `'42.828314258067e12'`

SRP

Include solar radiation pressure (SRP) — Acceleration due to solar radiation pressure

`off` (default) | `on`

To include acceleration due to solar radiation pressure in orbit propagation calculation, select this check box. Otherwise, clear the check box.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.

Programmatic Use

Block Parameter: `useSRP`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Eclipse fraction source — Source of eclipse fraction

`Dialog` (default) | `Port`

Source of the eclipse fraction (fraction of solar disk visible from the spacecraft location), specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.

Programmatic Use

Block Parameter: `useSRP`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

Shadow model — Shadow model

`Dual cone` (default) | `Cylindrical`

Shadow model for eclipse calculations, specified as one of these values.

- `Cylindrical` — Fraction can be 0.0 (Umbra) or 1.0 (Sunlight).
- `Dual cone` — Fraction can be 0.0 (Umbra), between 0.0 and 1.0 (Penumbra or Antumbra), or 1.0 (Sunlight).

Programmatic Use
Block Parameter: `shadowModel`
Type: character vector
Values: `'Dual cone'` | `'Cylindrical'`
Default: `'Dual cone'`

Include Earth — Option to include Earth

`on` (default) | `off`

Option to include Earth as a secondary occulting body in eclipse calculations when central body is Moon.

Dependencies

Set the **Central body** parameter to `Moon`.

Programmatic Use
Block Parameter: `includeEarth`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Include Moon — Option to include Moon

`on` (default) | `off`

Option to include Moon as a secondary occulting body in eclipse calculations when central body is Earth.

Dependencies

Set the **Central body** parameter to `Earth`.

Programmatic Use
Block Parameter: `includeMoon`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

Reflectivity coefficient source — Source for spacecraft coefficient of reflectivity

`'Dialog'` (default) | `'Port'`

Source for the spacecraft coefficient of reflectivity, specified as `Dialog` or `Port`.

Dependencies

To enable this check box:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Select the **Include solar radiation pressure (SRP)** check box.

Programmatic Use
Block Parameter: `reflectivityCoeffSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'on'`

Reflectivity coefficient — Spacecraft coefficient of reflectivity

`1.8` (default) | scalar | 2D array of size *numSat* | between `[1,2]`

Spacecraft coefficient of reflectivity used by solar radiation pressure calculation, specified as a scalar, 2D array of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.
- Set the **Reflectivity coefficient source** parameter to `Dialog`.

Programmatic Use
Block Parameter: `reflectivityCoeff`
Type: character vector
Values: `'1.8'` | scalar | 2D array of size *numSat* | between `[1,2]`
Default: `'1.8'`

SRP area source — Source for spacecraft solar radiation pressure area

`Dialog` (default) | `Port`

Source for the spacecraft solar radiation pressure (SRP) area, specified as `Dialog` or `Port`.

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.

Programmatic Use
Block Parameter: `srpAreaSrc`
Type: character vector
Values: `'Dialog'` | `'Port'`
Default: `'Dialog'`

SRP area — Cross section area of the spacecraft seen by the Sun

`1.0` (default) | scalar | 2D array of size *numSat*

Cross section area of the spacecraft seen by the Sun, specified as a scalar or as a 2D array of size *numSat*. *numSat* is the number of spacecraft.

Tunable: Yes

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.
- Set the **SRP area source** parameter to `Dialog`.

Programmatic Use

Block Parameter: `srpArea`
Type: character vector
Values: `'1.0'` | scalar | 2D array of size *numSat*
Default: `'1.0'`

Solar flux pressure (W*s/m^3) — Spacecraft solar flux pressure

`4.5344321e-6` (default) | scalar

Solar flux pressure, specified as a scalar.

Tunable: Yes

Dependencies

To enable this parameter:

- Set the **Propagation method** parameter to `Numerical (high precision)`.
- Set the **Include solar radiation pressure (SRP)** parameter to `on`.

Programmatic Use

Block Parameter: `fluxPressure`
Type: character vector
Values: `4.5344321e-6` | scalar
Default: `'4.5344321e-6'`

Units

Units — Parameter and port units

`Metric (m/s)` (default) | `Metric (km/s)` | `Metric (km/h)` | `English (ft/s)` | `English (kts)`

Parameter and port units, specified as shown here.

Units	Forces	Moment	Mass	Inertia	Distance	Velocity	Acceleration	Area	Density
`Metric (m/s)`	Newton	Newton-meter	Kilograms	Kilogram m^2	meters	meters/sec	meters/sec^2	m^2	kg/m^3, some density outputs 1/m^3
`Metric (km/s)`	Newton	Newton-meter	Kilograms	Kilogram m^2	kilometers	kilometers/sec	kilometers/sec^2	m^2	kg/m^3, some density outputs 1/m^3
`Metric (km/h)`	Newton	Newton-meter	Kilograms	Kilogram m^2	kilometers	kilometers/hour	kilometers/hour^2	m^2	kg/m^3, some density outputs 1/m^3
`English (ft/s)`	Pound-force	Foot-pound	Slugs	Slug ft^2	feet	feet/sec	feet/sec^2	feet^2	lbm/ft^3, some density outputs 1/ft^3
`English (kts)`	Pound-force	Foot-pound	Slugs	Slug ft^2	nautical mile	knots	knots/sec	feet^2	lbm/ft^3, some density outputs 1/ft^3

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (m/s)'` | `'Metric (km/s)'` | `'Metric (km/h)'` | `'English (ft/s)'` | `'English (kts)'`
Default: `'Metric (m/s)'`

Angle units — Angle units

`Degrees` (default) | `Radians`

Parameter and port units for angles, specified as `Degrees` or `Radians`.

Programmatic Use
Block Parameter: `angleUnits`
Type: character vector
Values: `'Degrees'` | `'Radians'`
Default: `'Degrees'`

Time format — Time format for start date and time output

`Julian date` (default) | `Gregorian`

Time format for **Start date/time (UTC Julian date)** and output port $\mathbf{t_{utc}}$, specified as `Julian date` or `Gregorian`.

Programmatic Use
Block Parameter: `timeFormat`
Type: character vector
Values: `'Julian date'` | `'Gregorian'`
Default: `'Julian date'`

Algorithms

Earth-Centric Coordinate Systems

The Spacecraft Dynamics block works in the ICRF and fixed-frame coordinate systems.

- ICRF — International Celestial Reference Frame. This frame can be treated as equal to the ECI coordinate system realized at J2000 (Jan 1 2000 12:00:00 TT). For more information, see “ECI Coordinates” on page 2-12.
- Fixed-frame — Fixed-frame is a generic term for the coordinate system that is fixed to the central body. The axes of the system rotate with the central body and are not fixed in inertial space. If the **Use Earth orientation parameters (EOPs)** check box is not selected, the block still uses the IAU2000/2005 reduction, but with Earth orientation parameters set to 0.

  - When **Central Body** is `Earth` and the **Use Earth orientation parameters (EOPs)** check box is selected, the fixed-frame coordinate system for the Moon is the Mean Earth/pole axis frame (ME). This frame is realized by two transformations. First, the values in the ICRF frame are transformed into the Principal Axis system (PA), which is the axis defined by the libration angles provided as inputs to the block (for more information, see Moon Libration). The states are then transformed into the ME system using a fixed rotation from the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [7].
  - When **Central Body** is `Moon` and the **Input Moon libration angles** check box is selected, the fixed-frame coordinate system for the Moon is the coordinate system defined by the libration angles provided as inputs to the block (for more information, see Moon Libration).
  - When **Central Body** is `Custom`, the fixed-frame coordinate system is defined by the poles of rotation and prime meridian defined by the block input α, δ, W, or the spin axis properties. In all other cases, the fixed frame for each central body is defined by the directions of the poles of rotation and prime meridians defined in the "Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006" [7].

Vehicle-Centric Coordinate Systems

The Spacecraft Dynamics block system works in the Body frame, north-east-down (NED), and local vertical, local horizontal (LVLH) coordinate systems.

- Body frame — Fixed in both origin and orientation to the moving craft. For more information, see “Body Coordinates” on page 2-9.
- North-east-down — Noninertial system with its origin fixed at the aircraft or spacecraft center of gravity. For more information, see “NED Coordinates” on page 2-11.
- Local vertical, local horizontal — Also known as the spacecraft coordinate system, Gaussian coordinate system, or the orbit frame. LVLH is a rotating, accelerating frame commonly used in studies of relative motion, such as vehicle maneuvering. The axes of this frame are:

- R-axis — Points outward from the spacecraft origin along its position vectors (with respect to the center of Earth). Measurements along this axis are referred to as radial.
- S-axis — Completes the right hand coordinate system. This axis points in the direction of the velocity vector, but is only parallel to it for circular orbits. Measurements along this axis are referred to as along-track or transverse.
- W-axis — Points normal to the orbital plane. Measurements along this axis are referred to as cross-track.

Translational Dynamics

The Spacecraft Dynamics block uses the Simulink solver to solve translational and rotational equations of motion of one or more spacecraft. The block translational dynamics are governed by these equations:

$$\overrightarrow{a_{icrf}} = \vec{a}_{centralbodygravity} + body2inertial\left(\frac{\vec{F}_b}{m}\right) + \overrightarrow{a_{applied}}$$

$$\overrightarrow{a_{icrf}} \xRightarrow[\text{integrate}]{} \overrightarrow{r_{icrf}}, \overrightarrow{v_{icrf}}$$

where:

- $\vec{a}_{applied}$ are the custom acceleration components from the **A** (applied acceleration) port.
- $\vec{F}_b$ are the input body force components.
- m is the spacecraft mass.

The method for computing central body acceleration depends on the current setting for the **Gravitational potential model** parameter. For gravity models that include nonspherical acceleration terms, the block computes nonspherical gravity in a fixed-frame coordinated system (for example, ITRF, in the case of Earth). However, the block always performs numerical integration in the inertial ICRF coordinate system. Therefore, at each timestep, the block:

1. Transforms position and velocity states into the fixed-frame.
2. Calculates nonspherical gravity in the fixed-frame.
3. Transforms the resulting acceleration into the inertial frame.
4. Sums the resulting acceleration with the other acceleration terms.
5. Integrates the summed acceleration terms.

Point-Mass

This option treats the central body as a point-mass, including only the effects of spherical gravity using Newton's law of universal gravitation.

$$\vec{a}_{\text{centralBodyGravity}} = -\frac{\mu}{r^2}\frac{\overrightarrow{r_{icrf}}}{r}$$

where μ is the standard gravitation parameter of the central body.

Oblate Ellipsoid

In addition to spherical gravity, this option includes the perturbing effects of the second-degree, zonal harmonic gravity coefficient J_2, accounting for the oblateness of the central body. J_2 accounts for the vast majority of the central bodies gravitational departure from a perfect sphere.

$$\vec{a}_{\mathrm{centralBodyGravity}} = -\frac{\mu}{r^2}\frac{\overrightarrow{r_{icrf}}}{r} + fixed2inertial\left(\vec{a}_{nonspherical}\right),$$

where:

$$\begin{aligned}\vec{a}_{nonspherical} = &\left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{ff_k}}{r^2\sqrt{r_{ff_i}{}^2 + r_{ff_j}{}^2}}\frac{\partial}{\partial\phi}U\right]r_{ff_i}\right\}i\\ &+\left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{ff_k}}{r^2\sqrt{r_{ff_i}{}^2 + r_{ff_j}{}^2}}\frac{\partial}{\partial\phi}U\right]r_{ff_j}\right\}j\\ &+\left\{\frac{1}{r}\left(\frac{\partial}{\partial r}U\right)r_k + \sqrt{\frac{r_{ff_i}{}^2 + r_{ff_j}{}^2}{r^2}}\frac{\partial}{\partial\phi}U\right\}k\end{aligned}$$

given the partial derivatives in spherical coordinates:

$$\frac{\partial}{\partial r}U = \frac{3\mu}{r^2}\left(\frac{R_{cb}}{r}\right)^2 P_{2,0}[\sin(\phi)]J_2$$

$$\frac{\partial}{\partial\phi}U = -\frac{u}{r}\left(\frac{R_{cb}}{r}\right)^2 P_{2,1}[\sin(\phi)]J_2$$

where:

- ϕ and λ are the satellite geocentric latitude and longitude.
- $P_{2,0}$ and $P_{2,1}$ associated Legendre functions.
- μ is the standard gravitation parameter of the central body.
- R_{cb} is the central body equatorial radius.

The transformation `fixed2inertial` converts fixed-frame position, velocity, and acceleration into the ICRF coordinate system with origin at the center of the central body, accounting for centrifugal and coriolis acceleration. For more information about the fixed and intertial coordinate systems used for each central body, see “Earth-Centric Coordinate Systems” on page 5-1204.

Spherical Harmonics

This option adds increased fidelity by including higher-order perturbation effects accounting for zonal, sectoral, and tesseral harmonics. For reference, the second-degree, zeroth order zonal harmonic J_2 is $-C_{2,0}$. The Spherical Harmonics model accounts for harmonics up to max degree $l=l_{\max}$, which varies by central body and geopotential model.

$$\vec{a}_{\mathrm{centralBodyGravity}} = -\frac{\mu}{r^2}\frac{\overrightarrow{r_{icrf}}}{r} + fixed2inertial\left(\vec{a}_{nonspherical}\right),$$

where

$$\vec{a}_{nonspherical} = \left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{ff_k}}{r^2\sqrt{r_{ff_i}{}^2 + r_{ff_j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{ff_i}\right\}i$$
$$+\left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{ff_k}}{r^2\sqrt{r_{ff_i}{}^2 + r_{ff_j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{ff_j}\right\}j$$
$$+\left\{\frac{1}{r}\left(\frac{\partial}{\partial r}U\right)r_k + \sqrt{\frac{r_{ff_i}{}^2 + r_{ff_j}{}^2}{r^2}}\frac{\partial}{\partial \phi}U\right\}k$$

given the partial derivatives

$$\frac{\partial}{\partial r}U = -\frac{u}{r^2}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{cb}}{r}\right)^l(l+1)P_{l,m}[\sin(\phi)]\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\}$$

$$\frac{\partial}{\partial \phi}U = \frac{u}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{cb}}{r}\right)^l\{P_{l,m+1}[\sin(\phi)] - (m)\tan(\phi)P_{l,m}[\sin(\phi)]\}\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\}$$

$$\frac{\partial}{\partial \lambda}U = \frac{u}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{cb}}{r}\right)^l(m)P_{l,m}[\sin(\phi)]\{S_{l,m}\cos(m\lambda) - C_{l,m}\sin(m\lambda)\},$$

where:

- ϕ and λ are the satellite geocentric latitude and longitude.
- $P_{l,m}$ are associated Legendre functions.
- μ is the standard gravitation parameter of the central body.
- R_{cb} is the central body equatorial radius.
- $C_{l,m}$ and $S_{l,m}$ are the unnormalized harmonic coefficients.

The transformation `fixed2inertial` converts fixed-frame position, velocity, and acceleration into the ICRF coordinate system with origin at the center of the central body, accounting for centrifugal and coriolis acceleration. For more information about the fixed and intertial coordinate systems used for each central body, see "Earth-Centric Coordinate Systems" on page 5-1204.

Rotational Dynamics

Rotational dynamics are governed by:

$$\dot{\vec{\omega}}_{b_{icrf}} = \left[\overrightarrow{M_b} - \vec{\omega}_{b_{icrf}} \times \left(I_{mom}\vec{\omega}_{b_{icrf}}\right) - \dot{I}_{mom}\vec{\omega}_{b_{icrf}}\right]\mathrm{inv}(I_{mom})$$

$$\dot{\vec{\omega}}_{b_{icrf}} \xRightarrow[\text{integrate}]{} \vec{q}_{b_{icrf}}, \vec{\omega}_{b_{icrf}},$$

where:

- $\overrightarrow{M_b}$ are the body moment components.
- I_{mom} is the spacecraft inertia tensor matrix.

When **Mass type** is `Fixed`, $\dot{I}_{mom}$ equals 0.

When **Mass type** is `Simple Variable`, this equation estimates the rate of change of the inertia tensor:

$$\dot{I}_{mom} = \frac{I_{full} - I_{empty}}{m_{full} - m_{empty}}\dot{m}$$

This equation gives the rate of change of the quaternion vector:

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} 0 & \omega_b(1) & \omega_b(2) & \omega_b(3) \\ -\omega_b(1) & 0 & -\omega_b(3) & \omega_b(2) \\ -\omega_b(2) & \omega_b(3) & 0 & -\omega_b(1) \\ -\omega_b(3) & -\omega_b(2) & \omega_b(1) & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Atmospheric Drag

The Spacecraft Dynamics block uses this atmospheric drag equation:

$$a_{drag} = -\frac{1}{2}\rho\left(\frac{C_D A}{m}\right)v^2{}_{rel}\frac{\vec{v}_{rel}}{v_{rel}}$$

where:

- m — Spacecraft mass used by atmospheric drag calculation.
- C_D — Coefficient of drag assuming that it is dimensionless.
- ρ — Atmospheric density.
- A — Area normal to v_{rel}, where

  $$\vec{v}_{rel} = \vec{v}_{sat} + \vec{v}_{atmos}$$
- v_{rel} — Velocity relative to atmosphere.

  $$\vec{v}_{rel} = \vec{v}_{icrf} - \vec{\omega}_{\oplus} \times \overrightarrow{r_{icrf}}$$

  where $\vec{\omega}_{\oplus}$ is the central body angular velocity.

Third Body

The Spacecraft Dynamics block uses this third body contribution equation:

$$a_{third\_body} = \mu_{third}\left(\frac{\vec{r}_{sat,3}}{r^3{}_{sat,3}} - \frac{\vec{r}}{r^3}\right)$$

where:

- μ_{third} — Gravitational parameter of the third body.
- $\vec{r}_{sat,3}$ — Vector from the satellite to the third body.
- $\vec{r}$ — Position of third body with regard to the central body, specified as a vector.

Solar Radiation Pressure

The Spacecraft Dynamics block uses this solar radiation pressure equation:

$$a_{srp} = -\upsilon C_r \frac{A_s}{m} P_{srp} \left(\frac{AU}{r_{sat,sun}} \right)^2 \frac{\vec{r}_{sat,sun}}{r_{sat,sun}}$$

where:

- m — Mass of the spacecraft.
- ν — Eclipse shadow function.
- C_r — Spacecraft coefficient of reflectivity.
- A_s — Spacecraft solar radiation pressure area.
- P_{srp} — Solar radiation pressure at a distance of AU from the sun.
- AU — Mean distance from the Sun to Earth (1AU).
- $|\vec{r}_{sat,sun}$ — Vector from the satellite to the Sun origin

Version History

Introduced in R2021b

R2023b: Spacecraft Dynamics Block Changes

The Spacecraft Dynamics block has been updated to take into account:

- The effects of third body gravity on orbit propagation.
- Space weather data from a data file generated by the `aeroReadSpaceWeatherData` function.
- Solar radiation pressure (SRP).

References

[1] Vallado, David. *Fundamentals of Astrodynamics and Applications*. 4th ed. Hawthorne, CA: Microcosm Press, 2013.

[2] Vepa, Ranjan. *Dynamics and Control of Autonomous Space Vehicles and Robotics*. New York: Cambridge University Press, 2019.

[3] Stevens, Frank L., and Brian L. Stevens. *Aircraft Control and Simulation*. 2nd ed. Hoboken, NJ: John Wiley & Sons, 2003.

[4] Gottlieb, R. G. *Fast Gravity, Gravity Partials, Normalized Gravity, Gravity Gradient Torque and Magnetic Field: Derivation, Code and Data*. NASA Contractor Report 188243. Houston: NASA, February 1993.

[5] Konopliv, A. S., S. W. Asmar, E. Carranza, W. L. Sjogen, D. N. Yuan. "Recent Gravity Models as a Result of the Lunar Prospector Mission." *Icarus* 150, no. 1 (2001): 1–18.

[6] Lemoine, F. G. et al. "An Improved Solution of the Gravity Field of Mars (GMM-2B) from Mars Global Surveyor." *Journal of Geophysical Research* 106, no. E10 (2001): 23359–23376.

[7] Seidelmann, P. Kenneth et al. "Report of the IAU/IAG Working Group on Cartographic Coordinates and Rotational Elements: 2006." *Celestial Mech Dyn Astr* 98 (20017): 155–180 (2007).

[8] Standish, E. M. "JPL Planetary and Lunar Ephemerides." DE405/LE405. Interoffice memorandum. JPL IOM 312.F-98-048. August 26, 1998.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

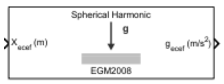
Orbit Propagator | CubeSat Vehicle | Moon Libration | Attitude Dynamics | Attitude Profile | Solar Flux and Geomagnetic Index | NRLMSISE-00 Atmosphere Model

Functions

`aeroReadSpaceWeatherData`

Spherical Harmonic Gravity Model

Implement spherical harmonic representation of planetary gravity



Libraries:
Aerospace Blockset / Environment / Gravity

Description

The Spherical Harmonic Gravity Model block implements the mathematical representation of spherical harmonic planetary gravity based on planetary gravitational potential. It provides a convenient way to describe a planet gravitational field outside of its surface in spherical harmonic expansion.

You can use spherical harmonics to modify the magnitude and direction of spherical gravity ($-GM/r^2$). The most significant or largest spherical harmonic term is the second degree zonal harmonic, J2, which accounts for oblateness of a planet.

Use this block if you want more accurate gravity values than spherical gravity models. For example, nonatmospheric flight applications might require higher accuracy.

Limitations

- The block excludes the centrifugal effects of planetary rotation, and the effects of a precessing reference frame.
- Spherical harmonic gravity model is valid for radial positions greater than the planet equatorial radius. Minor errors might occur for radial positions near or at the planetary surface. The spherical harmonic gravity model is not valid for radial positions less than the planetary surface.

Ports

Input

X_{ff} — Fixed-frame coordinates
N-by-3 matrix

Fixed-frame coordinates from center of planet, specified as an N-by-3 matrix, in selected units. Each row of the matrix is a separate position to calculate. The *z*-axis is positive toward the North Pole. If **Central body model** has a value of `EGM2008` or `EGM96`, this matrix contains Earth-centered Earth-fixed (ECEF) coordinates.

When inputting a large fixed-frame matrix and a high degree value, you might receive an out-of-memory error. For more information about avoiding out-of-memory errors in the MATLAB environment, see “Resolve “Out of Memory” Errors”.

When inputting a large fixed-frame matrix, you might receive a maximum matrix size limitation. To determine the largest matrix or array that you can create in the MATLAB environment for your platform, see “Performance and Memory”.

Data Types: double

Output

g_{ff} — Gravity values
N-by-3 matrix

Array of gravity values in the *x*-axis, *y*-axis, and *z*-axis of the fixed-frame coordinates, in selected length units per second squared. Each row of the matrix returns the calculated gravity vector for the corresponding row in the input matrix.

Data Types: double

Parameters

Units — Input and output units

Metric (MKS) (default) | English

Input and output units, specified as:

Units	Input	Output
Metric (MKS)	Meters (m)	Meters/sec^2 (m/s^2)
English	Feet (ft)	Feet/sec^2 (ft/s^2)

Programmatic Use
Block Parameter: units
Type: character vector
Values: 'Metric (MKS)' | 'English'
Default: 'Metric (MKS)'

Action for out-of-range input — Out-of-range input behavior

Warning (default) | Error | None

Out-of-range input behavior, specified as:

Value	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error	MATLAB returns an exception, model simulation stops.

The spherical harmonic gravity model is invalid for radial positions less than the planetary surface. The Spherical Harmonic Gravity Model block accepts out of range radial position inputs (less than planetary equatorial radius) when **Action for out-of-range input** is set to None or Warning. However, the block output might not be accurate or reliable.

Programmatic Use
Block Parameter: action
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'Warning'

Central body model — Planetary model

`EGM2008` (default) | `EGM96` | `LP100K` | `LP165P` | `GMM2B` | `Custom` | `EIGENGL04C`

Planetary model, specified as:

Central body model	Notes
`EGM2008`	Earth — Is the latest Earth spherical harmonic gravitational model from National Geospatial-Intelligence Agency (NGA). This block provides the WGS-84 version of this gravitational model. You can use the EGM96 planetary model if you need to use the older standard for Earth.
`EGM96`	Earth
`LP100K`	Moon — Is best for lunar orbit determination based upon computational time required to compute orbits. This planet model was created in approximately the same year as LP165P with similar data.
`LP165P`	Moon — Is best for extended lunar mission orbit accuracy. This planet model was created in approximately the same year as LP100K with similar data.
`GMM2B`	Mars
`Custom`	Enables you to specify your own planetary model. This option enables the **Central body MAT-file** parameter.
`EIGENGL04C`	Earth — Supports the gravity field model, EIGEN-GL04C (`http://icgem.gfz-potsdam.de/tom_longtime`). This model is an upgrade to EIGEN-CG03C.

For more information on the fixed-frame coordinate system for the central bodies, see “Algorithms” on page 5-1215.

When defining your own planetary model, the **Degree** parameter is limited to the maximum value for int16. When inputting a large degree, you might receive an out-of-memory error. For more information about avoiding out-of-memory errors in the MATLAB environment, see “Resolve “Out of Memory” Errors”.

Dependencies

Setting this parameter to `Custom` enables **Central body MAT-file**.

Programmatic Use

Block Parameter: `ptype`
Type: character vector
Values: `'EGM2008'` | `'EGM96'` | `'LP100K'` | `'LP165P'` | `'GMM2B'` | `'Custom'` | `'EIGENGL04C'`
Default: `'EGM2008'`

Degree — Degree of harmonic model

`120` (default) | scalar

Degree of harmonic model, specified as a scalar:

Central body model	Recommended Degree	Maximum Degree
EGM2008	120	2159
EGM96	70	360
LP100K	60	100
LP165P	60	165
GMM2B	60	80
EIGENGL04C	70	360

Programmatic Use
Block Parameter: `degree`
Type: character vector
Values: `scalar`
Default: `'120'`

Central body MAT-file — Central body MAT-file

`'aerogmm2b.mat'` (default)

Central body MAT-file that contains definitions for a custom planetary model. The `aerogmm2b.mat` file in Aerospace Blockset is the default MAT-file for a custom planetary model.

This file must contain:

Variable	Description
Re	Scalar of planet equatorial radius in meters (m).
GM	Scalar of planetary gravitational parameter in meters cubed per second squared (m^3/s^2)
degree	Scalar of maximum degree.
C	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *C*.
S	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *S*.

When using a large value for **Degree**, you might receive an out-of-memory error. For more information about avoiding out-of-memory errors in the MATLAB environment, see “Resolve “Out of Memory” Errors”.

Dependencies

To enable this parameter, set **Central body model** to `Custom`.

Programmatic Use
Block Parameter: `datafile`
Type: character vector
Values: `'aerogmm2b.mat'` | MAT-file
Default: `'aerogmm2b.mat'`

Algorithms

The Spherical Harmonic Gravity block works in the fixed-frame coordinate system for the central bodies:

- Earth — The fixed-frame coordinate system is the Earth-centered Earth-fixed (ECEF) coordinate system.
- Moon — The fixed-frame coordinate system is the Principal Axis system (PA), the orientation specified by JPL planetary ephemeris DE403.
- Mars — The fixed-frame coordinate system is defined by the directions of the poles of rotation and prime meridians defined in [14].

Version History

Introduced in R2010a

References

[1] Gottlieb, Robert G., "Fast Gravity, Gravity Partials, Normalized Gravity, Gravity Gradient Torque and Magnetic Field: Derivation, Code and Data." NASA-CR-188243. Houston, TX: NASA Lyndon B. Johnson Space Center, February 1993.

[2] Vallado, David. *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

[3] "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems." NIMA TR8350.2.

[4] Konopliv, A.S., W. Asmar, E. Carranza, W.L. Sjogren, and D.N. Yuan. "Recent Gravity Models as a Result of the Lunar Prospector Mission," *Icarus*, 150, no. 1 (2001): 1–18.

[5] Lemoine, F. G., D. E. Smith, D.D. Rowlands, M.T. Zuber, G. A. Neumann, and D. S. Chinn. "An Improved Solution of the Gravity Field of Mars (GMM-2B) from Mars Global Surveyor". *Journal Of Geophysical Research* 106, np E10 (October 25, 2001): pp 23359-23376.

[6] Kenyon S., J. Factor, N. Pavlis, and S. Holmes. "Towards the Next Earth Gravitational Model." Society of Exploration Geophysicists 77th Annual Meeting, San Antonio, TX, September 23–28, 2007.

[7] Pavlis, N.K., S.A. Holmes, S.C. Kenyon, and J.K. Factor, "An Earth Gravitational Model to Degree 2160: EGM2008." Presented at the 2008 General Assembly of the European Geosciences Union, Vienna, Austria, April 13–18, 2008.

[8] Grueber, T., and A. Köhl. "Validation of the EGM2008 Gravity Field with GPS-Leveling and Oceanographic Analyses." Presented at the IAG International Symposium on Gravity, Geoid & Earth Observation 2008, Chania, Greece, June 23–27, 2008.

[9] Förste, C., Flechtner et al, "A Mean Global Gravity Field Model From the Combination of Satellite Mission and Altimetry/Gravmetry Surface Data - EIGEN-GL04C." *Geophysical Research Abstracts* 8, 03462, 2006.

[10] Hill, K. A. "Autonomous Navigation in Libration Point Orbits." Doctoral dissertation, University of Colorado, Boulder. 2007.

[11] Colombo, Oscar L. "Numerical Methods for Harmonic Analysis on the Sphere." Reports of the Department of Geodetic Science, Report No. 310, The Ohio State University, Columbus, OH., March 1981.

[12] Colombo, Oscar L. "The Global Mapping of Gravity with Two Satellites." Netherlands Geodetic Commission 7, no 3, Delft, The Netherlands, 1984., Reports of the Department of Geodetic Science. Report No. 310. Columbus: Ohio State University, March 1981.

[13] Jones, Brandon A. "Efficient Models for the Evaluation and Estimation of the Gravity Field." Doctoral dissertation, University of Colorado, Boulder. 2010.

[14] Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 1991.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

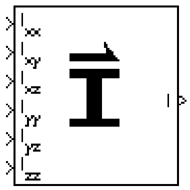
See Also

Centrifugal Effect Model | Zonal Harmonic Gravity Model

Topics
"Resolve "Out of Memory" Errors"

Symmetric Inertia Tensor

Create inertia tensor from moments and products of inertia



Libraries:
Aerospace Blockset / Mass Properties

Description

The Symmetric Inertia Tensor block creates an inertia tensor from moments and products of inertia. Each input corresponds to an element of the tensor.

The inertia tensor has the form of:

$$Inertia = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

Ports

Input

I_{xx} — Moment of inertia
scalar

Moment of inertia about the *x*-axis, specified as a scalar.

Data Types: `double`

I_{xy} — Product of inertia in *xy* plane
scalar

Product of inertia in the *xy* plane, specified as a scalar.

Data Types: `double`

I_{xz} — Product of inertia in *xz* plane
scalar

Product of inertia in the *xz* plane, specified as a scalar.

Data Types: `double`

I_{yy} — Moment of inertia about *y*-axis
scalar

Moment of inertia about the *y*-axis, specified as a scalar.

Data Types: `double`

I_{yz} — Product of inertia in *yz* plane
scalar

Product of inertia in the *yz* plane, specified as a scalar.

Data Types: `double`

I_{zz} — Moment of inertia about *z*-axis
scalar

Moment of inertia about the *z*-axis, specified as a scalar.

Data Types: `double`

Output

I — Inertia tensor
3-by-3 matrix

Symmetric inertia tensor, returned as a 3-by-3 matrix.

Data Types: `double`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

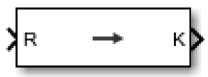
Generate C and C++ code using Simulink® Coder™.

See Also

Create 3x3 Matrix

Temperature Conversion

Convert from temperature units to desired temperature units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Temperature Conversion block computes the conversion factor from specified input temperature units to specified output temperature units and applies the conversion factor to the input signal.

The Temperature Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Temperature
scalar | array

Temperature, specified as a scalar or array, in initial temperature units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Temperature
scalar | array

Temperature, returned as a scalar or array, in final temperature units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

R (default) | F | C | K

Input units, specified as:

K	Kelvin
F	Degrees Fahrenheit
C	Degrees Celsius
R	Degrees Rankine

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use
Block Parameter: `IU`
Type: character vector
Values: `'K'` | `'F'` | `'C'` | `'R'`
Default: `'R'`

Final unit — Output units

K (default) | F | C | R

Output units, specified as:

K	Kelvin
F	Degrees Fahrenheit
C	Degrees Celsius
R	Degrees Rankine

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use
Block Parameter: `OU`
Type: character vector
Values: `'K'` | `'F'` | `'C'` | `'R'`
Default: `'K'`

Version History
Introduced before R2006a

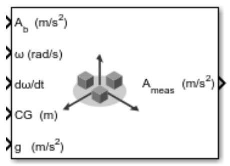
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Velocity Conversion

Three-axis Accelerometer

Implement three-axis accelerometer



Libraries:
Aerospace Blockset / GNC / Navigation

Description

The Three-Axis Accelerometer block implements an accelerometer on each of the three axes. For more information on the ideal measured accelerations, see “Algorithms” on page 5-1226.

Optionally, to apply discretizations to the Three-Axis Accelerometer block inputs and dynamics along with nonlinearizations of the measured accelerations, use the Saturation block.

The Three-axis Accelerometer block icon displays the input and output units selected from the **Units** parameter.

Limitations

- Vibropendulous error and hysteresis effects are not accounted for in this block.
- This block is not intended to model the internal dynamics of different forms of the instrument.

Ports

Input

A_b — Actual accelerations
three-element vector

Actual accelerations in body-fixed axes, specified as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

ω — Angular rates
three-element vector

Angular rates in body-fixed axes, specified as a three-element vector, in radians per second.

Data Types: `double`

dω/dt — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, specified as a three-element vector, in radians per second squared.

Data Types: `double`

CG — Location of center of gravity
three-element vector

Location of the center of gravity, specified as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

g — Gravity
three-element vector

Gravity in body axis, specified as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

Output

A_{meas} — Measured accelerations
three-element vector

Measured accelerations from the accelerometer, returned as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Input and output units, specified as:

Units	Acceleration	Length
`Metric (MKS)`	Meters per second squared	Meters
`English` (British Imperial)	Feet per second squared	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Accelerometer location — Accelerometer location

`[0 0 0]` (default) | three-element vector

Location of the accelerometer group, specified as a three-element vector, measured from the zero datum (typically the nose) to aft, to the right of the vertical centerline, and above the horizontal centerline. This measurement reference is the same for the center of gravity input. The units are the units specified in the **Units** parameter.

Programmatic Use
Block Parameter: `acc`
Type: character vector
Values: three-element vector
Default: `'[0 0 0]'`

Subtract gravity — Subtract gravity

`on` (default) | `off`

To subtract gravity from acceleration readings, select this check box.

Programmatic Use
Block Parameter: `gtype`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Second order dynamics — Second-order dynamics

`on` (default) | `off`

To apply second-order dynamics to acceleration readings, select this check box.

Programmatic Use
Block Parameter: `dtype_a`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Natural frequency (rad/sec) — Natural frequency

`190` (default) | scalar

Natural frequency of the accelerometer, specified as a double scalar, in radians per second.

Programmatic Use
Block Parameter: `w_a`
Type: character vector
Values: double scalar
Default: `'190'`

Damping ratio — Damping ratio

`0.707` (default) | scalar

Damping ratio of the accelerometer, specified as a double scalar, with no dimensions.

Programmatic Use
Block Parameter: `z_a`
Type: character vector
Values: double scalar
Default: `'0.707'`

Scale factors and cross-coupling — Scale factors and cross coupling

`[1 0 0; 0 1 0; 0 0 1]` (default) | 3-by-3 matrix

Scale factors and cross-coupling, specified as a 3-by-3 matrix, to skew the accelerometer from body axes and to scale accelerations along body axes.

Programmatic Use
Block Parameter: `a_sf_cc`
Type: character vector
Values: 3-by-3 matrix
Default: `'[1 0 0; 0 1 0; 0 0 1]'`

Measurement bias — Measurement bias

`[0 0 0]` (default) | three-element vector

Long-term biases along the accelerometer axes, specified as a three-element vector, in the units specified in the **Units** parameter.

Programmatic Use
Block Parameter: `a_bias`
Type: character vector
Values: 3-by-3 matrix
Default: `'[0 0 0]'`

Update rate (sec) — Update rate

`0` (default) | scalar

Update rate of the accelerometer, specified as a double scalar, in seconds. An update rate of 0 creates a continuous accelerometer. If the **Noise on** check box is selected and the update rate is 0, the block updates the noise at a rate of 0.1.

Tip If you:

- Update this parameter value to 0 (continuous)
- Configure a fixed-step solver for the model

you must also select the **Automatically handle rate transition for data transfer** check box in the **Solver** pane. This check box enables the software to handle rate transitions correctly.

Programmatic Use
Block Parameter: `a_Ts`
Type: character vector
Values: double scalar
Default: `'0'`

Noise on — White noise

on (default) | `off`

To apply white noise to acceleration readings, select this check box.

Programmatic Use

Block Parameter: `a_rand`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Noise seeds — Noise seeds

`[23093 23094 23095]` (default) | three-element vector

Scalar seeds for the Gaussian noise generator for each axis of the accelerometer, specified as a three-element vector.

Dependencies

To enable this parameter, select **Noise on**.

Programmatic Use

Block Parameter: `a_seeds`
Type: character vector
Values: three-element vector
Default: `'[23093 23094 23095]'`

Noise power — Noise power

`[0.001 0.001 0.001]` (default) | three-element vector

Height of the power spectral density (PSD) of the white noise for each axis of the accelerometer, specified as a three-element vector, in:

- $(m/s^2)/Hz$ when **Units** is set to `Metric (MKS)`
- $(ft/s^2)/Hz$ when **Units** is set to `English`

Dependencies

To enable this parameter, select **Noise on**.

Programmatic Use

Block Parameter: `a_pow`
Type: character vector
Values: three-element vector
Default: `'[0.001 0.001 0.001]'`

Lower and upper output limits — Minimum and maximum values of acceleration

`[-inf -inf -inf inf inf inf]` (default) | six-element vector

Three minimum values and three maximum values of acceleration in each of the accelerometer axes, specified as a six-element vector, in the units specified in the **Units** parameter.

Programmatic Use

Block Parameter: `a_sat`
Type: character vector
Values: six-element vector
Default: `'[-inf -inf -inf inf inf inf]'`

Algorithms

The ideal measured accelerations ($\bar{A}_{imeas}$) include the acceleration in body axes at the center of gravity ($\bar{A}_b$) and lever arm effects due to the accelerometer not being at the center of gravity. Optionally, gravity in body axes can be removed. This is represented by the equation:

$$\bar{A}_{imeas} = \bar{A}_b + \bar{\omega}_b \times (\bar{\omega}_b \times \bar{d}) + \dot{\bar{\omega}}_b \times \bar{d} - \bar{g}$$

where $\bar{\omega}_b$ are body-fixed angular rates, $\dot{\bar{\omega}}_b$ are body-fixed angular accelerations, and $\bar{d}$ is the lever arm. The lever arm ($\bar{d}$) is defined as the distances that the accelerometer group is forward, right, and below the center of gravity:

$$\bar{d} = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} = \begin{bmatrix} -(x_{acc} - x_{CG}) \\ y_{acc} - y_{CG} \\ -(z_{acc} - z_{CG}) \end{bmatrix}$$

The orientation of the axes used to determine the location of the accelerometer group (x_{acc}, y_{acc}, z_{acc}) and center of gravity (x_{CG}, y_{CG}, z_{CG}) is from the zero datum (typically the nose) to aft, to the right of the vertical centerline and above the horizontal centerline. The x-axis and z-axis of these measurement axes are opposite the body-fixed axes that produce the negative signs in the lever arms for the x-axis and z-axis.

Measured accelerations ($\bar{A}_{meas}$) output by this block contain error sources and are defined as

$$\bar{A}_{meas} = \bar{A}_{imeas} \times \bar{A}_{SFCC} + \bar{A}_{bias} + noise,$$

where $\bar{A}_{SFCC}$ is a 3-by-3 matrix of scaling factors on the diagonal and misalignment terms in the nondiagonal, and $\bar{A}_{bias}$ are the biases.

Version History

Introduced before R2006a

References

[1] Rogers, R. M., *Applied Mathematics in Integrated Navigation Systems*, AIAA Education Series, 2000.

Extended Capabilities

C/C++ Code Generation

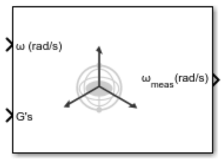
Generate C and C++ code using Simulink® Coder™.

See Also

Three-axis Gyroscope | Three-axis Inertial Measurement Unit | Saturation

Three-axis Gyroscope

Implement three-axis gyroscope



Libraries:
Aerospace Blockset / GNC / Navigation

Description

The Three-Axis Gyroscope block implements a gyroscope on each of the three axes. For more information on the measured body angular rates, see “Algorithms” on page 5-1230.

Optionally, to apply discretizations to the block inputs and dynamics along with nonlinearizations of the measured body angular rates, use the Saturation block.

Limitations

- Anisoelastic bias and anisoinertial bias effects are not accounted for in this block.
- This block is not intended to model the internal dynamics of different forms of the instrument.

Ports

Input

ω — Angular rates
three-element vector

Angular rates in the body-fixed axes, specified as a three-element vector, in radians per second.

Data Types: `double`

G's — Accelerations
three-element vector

Accelerations in the body-fixed axes, specified as a three-element vector, in Gs.

Data Types: `double`

Output

ω_{meas} — Measured angular rates
three-element vector

Measured angular rates from the gyroscope, returned as a three-element vector, in radians per second.

Data Types: `double`

Parameters

Second order dynamics — Second-order dynamics

on (default) | off

To apply second-order dynamics to gyroscope readings, select this check box.

Programmatic Use
Block Parameter: dtype_g
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Natural frequency (rad/sec) — Natural frequency

190 (default) | scalar

Natural frequency of the gyroscope, specified as a double scalar, in radians per second.

Programmatic Use
Block Parameter: w_g
Type: character vector
Values: double scalar
Default: '190'

Damping ratio — Damping ratio

0.707 (default) | scalar

Damping ratio of the gyroscope, specified as a double scalar.

Programmatic Use
Block Parameter: z_g
Type: character vector
Values: double scalar
Default: '0.707'

Scale factors and cross-coupling — Scale factors and cross coupling

[1 0 0; 0 1 0; 0 0 1] (default) | 3-by-3 matrix

Scale factors and cross-coupling, specified as a 3-by-3 matrix, to skew the gyroscope from body axes and to scale accelerations along body axes.

Programmatic Use
Block Parameter: g_sf_cc
Type: character vector
Values: 3-by-3 matrix
Default: '[1 0 0; 0 1 0; 0 0 1]'

Measurement bias — Measurement bias

[0 0 0] (default) | three-element vector

Long-term biases along the gyroscope axes, specified as a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `g_bias`
Type: character vector
Values: 3-by-3 matrix
Default: `'[0 0 0]'`

G-sensitive bias — Maximum change in rates

`[0 0 0]` (default) | three-element vector

Maximum change in rates due to linear acceleration, specified as a three-element vector, in radians per second per g-unit.

Programmatic Use
Block Parameter: `g_sen`
Type: character vector
Values: three-element vector
Default: `'[0 0 0]'`

Update rate (sec) — Update rate

`0` (default) | scalar

Update rate of the gyroscope, specified as a double scalar, in seconds. An update rate of 0 creates a continuous gyroscope. If the **Noise on** check box is selected and the update rate is 0, the block updates the noise at a rate of 0.1.

Tip If you:

- Update this parameter value to 0 (continuous)
- Configure a fixed-step solver for the model

you must also select the **Automatically handle rate transition for data transfer** check box in the **Solver** pane. This check box enables the software to handle rate transitions correctly.

Programmatic Use
Block Parameter: `g_Ts`
Type: character vector
Values: double scalar
Default: `'0'`

Noise on — White noise

`on` (default) | `off`

To apply white noise to the gyroscope readings, select this check box.

Programmatic Use
Block Parameter: `g_rand`
Type: character vector

Values: `'on'` | `'off'`
Default: `'on'`

Noise seeds — Noise seeds

`[23093 23094 23095]` (default) | three-element vector

Scalar seeds for the Gaussian noise generator for each axis of the gyroscope, specified as a three-element vector.

Dependencies

To enable this parameter, select **Noise on**.

Programmatic Use
Block Parameter: `g_seeds`
Type: character vector
Values: three-element vector
Default: `'[23093 23094 23095]'`

Noise power — Noise power

`[0.0001 0.0001 0.0001]` (default) | three-element vector

Height of the power spectral density (PSD) of the white noise for each axis of the gyroscope, specified as a three-element vector, in $(rad/s)^2/Hz$.

Dependencies

To enable this parameter, select **Noise on**.

Programmatic Use
Block Parameter: `g_pow`
Type: character vector
Values: three-element vector
Default: `'[0.0001 0.0001 0.0001]'`

Lower and upper output limits — Minimum and maximum values of angular rates

`[-inf -inf -inf inf inf inf]` (default) | six-element vector

Three minimum values and three maximum values of angular rates in each of gyroscope axes, specified as a six-element vector, in radians per second.

Programmatic Use
Block Parameter: `g_sat`
Type: character vector
Values: six-element vector
Default: `'[-inf -inf -inf inf inf inf]'`

Algorithms

The measured body angular rates ($\bar{\omega}_{meas}$) include the body angular rates ($\bar{\omega}_b$), errors, and, optionally, the discretizations and nonlinearizations of the signals:

$$\bar{\omega}_{meas} = \bar{\omega}_b \times \bar{\omega}_{SFCC} + \bar{\omega}_{bias} + Gs \times \bar{\omega}_{gsens} + noise$$

where $\bar{\omega}_{SFCC}$ is a 3-by-3 matrix of scaling factors on the diagonal and misalignment terms in the nondiagonal, $\bar{\omega}_{bias}$ are the biases, (Gs) are the Gs on the gyroscope, and $\bar{\omega}_{gsens}$ are the G-sensitive biases.

Version History

Introduced before R2006a

References

[1] Rogers, R. M., *Applied Mathematics in Integrated Navigation Systems*, AIAA Education Series, 2000.

Extended Capabilities

C/C++ Code Generation

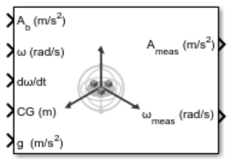
Generate C and C++ code using Simulink® Coder™.

See Also

Three-axis Accelerometer | Three-axis Inertial Measurement Unit | Saturation

Three-axis Inertial Measurement Unit

Implement three-axis inertial measurement unit (IMU)



Libraries:
Aerospace Blockset / GNC / Navigation

Description

The Three-Axis Inertial Measurement Unit block implements an inertial measurement unit (IMU) containing a three-axis accelerometer and a three-axis gyroscope.

For a description of the equations and application of errors, see Three-axis Accelerometer and Three-axis Gyroscope.

The Three-axis Inertial Measurement Unit block icon displays the input and output units selected from the **Units** parameter.

Limitations

- Vibropendulous error, hysteresis affects, anisoelastic bias, and anisoinertial bias are not accounted for in this block.
- This block is not intended to model the internal dynamics of different forms of the instrument.

Ports

Input

A_b — Actual accelerations
three-element vector

Actual accelerations in body-fixed axes, specified as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

ω — Angular rates
three-element vector

Angular rates in body-fixed axes, specified as a three-element vector, in radians per second.

Data Types: `double`

dω/dt — Angular accelerations
three-element vector

Angular accelerations in body-fixed axes, specified as a three-element vector, in radians per second squared.

Data Types: `double`

CG — Location of center of gravity
three-element vector

Location of the center of gravity, specified as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

g — Gravity
three-element vector

Gravity in body axis, specified as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

Output

A_{meas} — Measured accelerations
three-element vector

Measured accelerations from the accelerometer, returned as a three-element vector, in the units specified in the **Units** parameter.

Data Types: `double`

ω_{meas} — Measured angular rates
three-element vector

Measured angular rates from the gyroscope, returned as a three-element vector, in radians per second.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English`

Input and output units, specified as:

Units	Acceleration	Length
`Metric (MKS)`	Meters per second squared	Meters
`English` (British Imperial)	Feet per second squared	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

IMU location — IMU location

`[0 0 0]` (default) | three-element vector

The location of the IMU, which is also the accelerometer group location, is measured from the zero datum (typically the nose) to aft, to the right of the vertical centerline, and above the horizontal centerline. This measurement reference is the same for the center of gravity input. The units are in the units specified in the **Units** parameter.

Programmatic Use
Block Parameter: `imu`
Type: character vector
Values: three-element vector
Default: `'[0 0 0]'`

Update rate (sec) — Update rate

`0` (default) | scalar

Update rate of the accelerometer and gyroscope, specified as a double scalar, in seconds. An update rate of 0 creates a continuous accelerometer and continuous gyroscope. If the **Noise on** check box is selected and the update rate is 0, the block updates the noise at a rate of 0.1.

Tip If you:

- Update this parameter value to 0 (continuous)
- Configure a fixed-step solver for the model

you must also select the **Automatically handle rate transition for data transfer** check box in the **Solver** pane. This check box enables the software to handle rate transitions correctly.

Programmatic Use
Block Parameter: `i_Ts`
Type: character vector
Values: double scalar
Default: `'0'`

Second-order dynamics for accelerometer — Second-order dynamics

`on` (default) | `off`

To apply second-order dynamics to acceleration readings, select this check box.

Programmatic Use
Block Parameter: `dtype_a`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Accelerometer natural frequency (rad/sec) — Accelerometer natural frequency

`190` (default) | scalar

Natural frequency of the accelerometer, specified as a double scalar, in radians per second.

Dependencies

To enable this parameter, select **Second order dynamics for accelerometer**.

Programmatic Use
Block Parameter: `w_a`
Type: character vector
Values: double scalar
Default: `'190'`

Accelerometer damping ratio — Accelerometer damping ratio

`0.707` (default) | scalar

Damping ratio of the accelerometer, specified as a double scalar, with no dimensions.

Dependencies

To enable this parameter, select **Second order dynamics for accelerometer**.

Programmatic Use
Block Parameter: `z_a`
Type: character vector
Values: double scalar
Default: `'0.707'`

Accelerometer scale factor and cross-coupling — Scale factors and cross coupling

`[1 0 0; 0 1 0; 0 0 1]` (default) | 3-by-3 matrix

Scale factors and cross-coupling, specified as a 3-by-3 matrix, to skew the accelerometer from body axes and to scale accelerations along body axes.

Programmatic Use
Block Parameter: `a_sf_cc`
Type: character vector
Values: 3-by-3 matrix
Default: `'[1 0 0; 0 1 0; 0 0 1]'`

Accelerometer measurement bias — Accelerometer measurement bias

`[0 0 0]` (default) | three-element vector

Long-term biases along the accelerometer axes, specified as a three-element vector, in the units specified in the **Units** parameter.

Programmatic Use
Block Parameter: `a_bias`
Type: character vector
Values: three-element vector
Default: `'[0 0 0]'`

Accelerometer upper and lower limits — Minimum and maximum values of acceleration

`[-inf -inf -inf inf inf inf]` (default) | six-element vector

Three minimum values and three maximum values of acceleration in each of accelerometer axes, specified as a six-element vector, in units specified in the **Units** parameter.

Programmatic Use
Block Parameter: `a_sat`
Type: character vector
Values: six-element vector
Default: `'[-inf -inf -inf inf inf inf]'`

Second-order dynamics for gyro — Gyroscope second-order dynamics

`on` (default) | `off`

To apply second-order dynamics to gyroscope readings, select this check box.

Programmatic Use
Block Parameter: `dtype_g`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Gyro natural frequency (rad/sec) — Gyroscope natural frequency

`190` (default) | scalar

Natural frequency of the gyroscope, specified as a double scalar, in radians per second.

Dependencies

To enable this parameter, select **Second-order dynamics for gyro**.

Programmatic Use
Block Parameter: `w_g`
Type: character vector
Values: double scalar
Default: `'190'`

Gyro damping ratio — Gyroscope damping ratio

`0.707` (default) | scalar

Damping ratio of the gyroscope, specified as a double scalar.

Dependencies

To enable this parameter, select **Second-order dynamics for gyro**.

Programmatic Use
Block Parameter: `z_g`
Type: character vector
Values: double scalar
Default: `'0.707'`

Gyro scale factors and cross-coupling — Gyroscope scale factors and cross-coupling

`[1 0 0; 0 1 0; 0 0 1]` (default) | 3-by-3 matrix

Gyroscope scale factors and cross-coupling, specified as a 3-by-3 matrix, to skew the gyroscope from body axes and to scale angular rates along body axes.

Programmatic Use
Block Parameter: `g_sf_cc`
Type: character vector
Values: 3-by-3 matrix
Default: `'[1 0 0; 0 1 0; 0 0 1]'`

Gyro measurement bias — Gyroscope measurement bias

`[0 0 0]` (default) | three-element vector

Long-term biases along the gyroscope axes, specified a three-element vector, in radians per second.

Programmatic Use
Block Parameter: `g_bias`
Type: character vector
Values: three-element vector
Default: `'[0 0 0]'`

G-sensitive bias — Maximum change in rates

`[0 0 0]` (default) | three-element vector

Maximum change in rates due to linear acceleration, specified as a three-element vector, in radians per second per g-unit.

Programmatic Use
Block Parameter: `g_sens`
Type: character vector
Values: three-element vector
Default: `'[0 0 0]'`

Gyro upper and lower limits — Minimum and maximum values of angular rates

`[-inf -inf -inf inf inf inf]` (default) | six-element vector

Three minimum values and three maximum values of angular rates in each of the gyroscope axes, specified as a six-element vector, in radians per second.

Programmatic Use
Block Parameter: `g_sat`
Type: character vector
Values: six-element vector
Default: `'[-inf -inf -inf inf inf inf]'`

Noise on — White noise

on (default) | `off`

To apply white noise to acceleration and gyroscope readings, select this check box.

Programmatic Use
Block Parameter: `i_rand`
Type: character vector

Values: 'on' | 'off'
Default: 'on'

Noise seeds — Noise seeds

[23093 23094 23095 23096 23097 23098] (default) | six-element vector

Scalar seeds for the Gaussian noise generator for each axis of the accelerometer and gyroscope, specified as a six-element vector.

Dependencies

To enable this parameter, select **Noise on**.

Programmatic Use

Block Parameter: i_seeds
Type: character vector
Values: six-element vector
Default: '[23093 23094 23095 23096 23097 23098]'

Noise power — Noise power

[0.001 0.001 0.001 0.0001 0.0001 0.0001] (default) | six-element vector

Height of the power spectral density (PSD) of the white noise for each axis of the accelerometer and gyroscope, specified as a six-element vector, in:

- (m/s^2)/Hz when **Units** is set to Metric (MKS)
- (ft/s^2)/Hz when **Units** is set to English

Dependencies

To enable this parameter, select **Noise on**.

Programmatic Use

Block Parameter: i_pow
Type: character vector
Values: six-element vector
Default: '[0.001 0.001 0.001 0.0001 0.0001 0.0001]'

Version History

Introduced before R2006a

References

[1] Rogers, R. M., *Applied Mathematics in Integrated Navigation Systems*, AIAA Education Series, 2000.

Extended Capabilities

C/C++ Code Generation

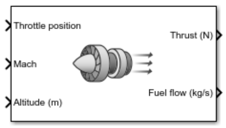
Generate C and C++ code using Simulink® Coder™.

See Also

Three-axis Accelerometer | Three-axis Gyroscope | Calculate Range

Turbofan Engine System

Implement first-order representation of turbofan engine with controller



Libraries:
Aerospace Blockset / Propulsion

Description

The Turbofan Engine System block computes the thrust and the fuel mass flow rate of a turbofan engine and controller at a specific throttle position, Mach number, and altitude. For more information on this system, see "Algorithms" on page 5-1243.

The Turbofan Engine System block icon displays the input and output units selected from the **Units** parameter.

Limitations

- The atmosphere is at standard day conditions and an ideal gas.
- The Mach number is limited to less than 1.0.
- This engine system is for indication purposes only. It is not meant to be used as a reference model.
- This engine system is assumed to have a high bypass ratio.

Ports

Input

Throttle position — Throttle position
scalar | vector

Throttle position, specified as a scalar or vector. This value can vary from zero to one, corresponding to no and full throttle.

Data Types: `double`

Mach — Mach number
scalar

Mach number, specified as a scalar.

Data Types: `double`

Altitude — Altitude
scalar | vector

Altitude, specified as scalar or vector, in the units specified in the **Units** parameter.

Data Types: `double`

Initial thrust — Initial thrust
scalar | vector

Initial thrust, specified as a scalar or vector, in the units specified in the **Units** parameter.

Dependencies

To enable this port, set **Initial thrust source** to `External`.

Data Types: `double`

Output

Thrust — Thrust
scalar | vector

Thrust, returned as a scalar or vector, in the units specified in the **Units** parameter.

Data Types: `double`

Fuel flow — Fuel flow
scalar | vector

Fuel flow, returned as scalar or vector, in the units specified in the **Units** parameter units per second.

Data Types: `double`

Parameters

Units — Input and output units

`Metric (MKS)` (default) | `English`

Input and output units, specified as:

Units	Altitude	Thrust	Fuel Flow
`Metric (MKS)`	Meters	Newtons	Kilograms per second
`English`	Feet	Pound force	Pound mass per second

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Initial thrust source — Initial thrust source

`Internal` (default) | `External`

Initial thrust, specified as:

`Internal`	Use the value of the **Initial thrust** parameter.
`External`	Use external input for initial thrust value.

Programmatic Use
Block Parameter: `ic_source`
Type: character vector
Values: `'Internal'` | `'External'`
Default: `'Internal'`

Initial thrust — Initial

`0` (default) | scalar

Initial thrust value, specified as a double scalar.

Programmatic Use
Block Parameter: `IC`
Type: character vector
Values: double scalar
Default: `'0'`

Maximum sea-level static thrust — Maximum thrust at sea-level

`45000` (default) | scalar

Maximum thrust at sea-level, specified as a double scalar, at a Mach value of 0.

Programmatic Use
Block Parameter: `Fmax`
Type: character vector
Values: double scalar
Default: `'45000'`

Fastest engine time constant at sea-level static (sec) — Fastest engine time at sea level

`1` (default) | scalar

Fastest engine time at sea level, specified as a double scalar.

Programmatic Use
Block Parameter: `tau`
Type: character vector
Values: double scalar
Default: `'1'`

Sea-level static thrust specific fuel consumption — Thrust-specific fuel consumption at sea level

`0.35` (default) | scalar

Thrust-specific fuel consumption at sea level, specified as a double scalar, in specified mass units per hour per specified thrust units, at:

- Mach value of 0
- Maximum thrust

Programmatic Use
Block Parameter: `SFC`
Type: character vector

Values: double scalar
Default: `'0.35'`

Ratio of installed thrust to uninstalled thrust — Coefficient representing loss

`0.9` (default) | scalar

Coefficient representing the loss in thrust due to engine installation, specified as a double value.

Programmatic Use
Block Parameter: `Nt`
Type: character vector
Values: double scalar
Default: `'0.9'`

Algorithms

This system is represented by a first-order system with unitless heuristic lookup tables for thrust, thrust specific fuel consumption (TSFC), and the engine time constant. For the lookup table data, thrust is a function of throttle position and the Mach number, TSFC is a function of thrust and the Mach number, and engine time constant is a function of thrust. The unitless lookup table outputs are corrected for altitude using the relative pressure ratio δ and relative temperature ratio θ, and scaled by maximum sea level static thrust, the fastest engine time constant at the sea level static, sea level static thrust specific fuel consumption, and the ratio of installed thrust to uninstalled thrust.

Version History

Introduced before R2006a

References

[1] *Aeronautical Vestpocket Handbook*, United Technologies Pratt & Whitney, August, 1986.

[2] Raymer, D. P., *Aircraft Design: A Conceptual Approach*, AIAA Education Series, Washington, DC, 1989.

[3] Hill, P. G., and C. R. Peterson, *Mechanics and Thermodynamics of Propulsion,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1970.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Turn Coordinator

Display measurements on turn coordinator and inclinometer

Libraries:
Aerospace Blockset / Flight Instruments

Description

The Turn Coordinator block displays measurements on a gyroscopic turn rate instrument and on an inclinometer.

- The gyroscopic turn rate instrument shows the rate of heading change of the aircraft as a tilting of the aircraft symbol in the gauge.
- The inclinometer shows whether the turn is coordinated, slipping, or skidding by the position of the ball.

When the ball is centered, the turn is coordinated. When the ball is off center, the turn is slipping or skidding. The turn rate instrument has marks for wings level and for a standard rate turn. A standard rate turn is a heading change of 3 degrees per second, also known as a two minute turn.

The input for gyroscopic turn rate instruments and inclinometers is in degrees. The turn rate value is input as the degrees of tilt of the aircraft symbol in the gauge. The standard rate turn marks are at angles of ±15 degrees. Tilt angle values are limited to ±20 degrees, whereas inclinometer angles are limited to ±15 degrees.

Combine the turn indicator and inclinometer signals in a Mux block in order:

1. Turn indicator
2. Inclinometer

For example, turn indicator and inclinometer values of `[15 0]` indicate a coordinated, standard rate turn.

Tip To facilitate understanding and debugging your model, you can modify instrument block connections in your model during normal and accelerator mode simulations.

Parameters

Connection — Connect to signal
signal name | 2-element signal

Connect to 2-element signal for display, selected from a list of signal names. The 2-element signal consists of turn indicator and inclinometer signals combined in a Mux block, in degrees. You connect and display this combined signal. This input cannot be a bus signal.

To view the data from a signal, select a signal in the model. The signal appears in the **Connection** table. Select the option button next to the signal you want to display. Click **Apply** to connect the signal.

The table has a row for the signal connected to the block. If there are no signals selected in the model, or the block is not connected to any signals, the table is empty.

Label — Block label location

`Top` (default) | `Bottom` | `Hide`

Block label, displayed at the top or bottom of the block, or hidden.

- `Top`

  Show label at the top of the block.
- `Bottom`

  Show label at the bottom of the block.
- `Hide`

  Do not show the label or instructional text when the block is not connected.

Programmatic Use
Block Parameter: `LabelPosition`
Type: character vector
Values: `'Top'` | `'Bottom'` | `'Hide'`
Default: `'Top'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

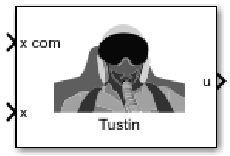
Airspeed Indicator | Altimeter | Artificial Horizon | Climb Rate Indicator | Exhaust Gas Temperature (EGT) Indicator | Heading Indicator | Revolutions Per Minute (RPM) Indicator

Topics

"Display Measurements with Cockpit Instruments" on page 2-59
"Flight Instrument Gauges" on page 2-58

Tustin Pilot Model

Represent Tustin pilot model



Libraries:
Aerospace Blockset / Pilot Models

Description

The Tustin Pilot Model block represents the pilot model that A. Tustin describes in *The Nature of the Operator's Response in Manual Control, and its Implications for Controller Design* [1]. When modeling human pilot models, use this block for the least accuracy, compared to that provided by the Crossover Pilot Model and Precision Pilot Model blocks. This block requires less input than those blocks, and provides better performance. However, the results might be less accurate.

This pilot model is a single input, single output (SISO) model that represents human behavior, and is based on the transfer function described in "Algorithms" on page 5-1247.

This block has nonlinear behavior. If you want to linearize the block (for example, with one of the `linmod` functions), you might need to change the Pade approximation order. The Tustin Pilot Model block implementation incorporates the Transport Delay block with the **Pade order (for linearization)** parameter set to `2` by default. To change this value, use the `set_param` function, for example:

```
set_param(gcb,'pade','3')
```

Ports

Input

x com — Signal command
scalar

Signal command that the pilot model controls, specified as a scalar.

Data Types: `double`

x — Signal controlled by pilot
scalar

Signal controlled by pilot, specified as a scalar.

Data Types: `double`

Output

u — Aircraft command
scalar

Aircraft command, returned as a scalar.

Data Types: `double`

Parameters

Pilot gain — Pilot gain

`1` (default) | scalar

Pilot gain, specified as a double scalar.

Programmatic Use
Block Parameter: `Kp`
Type: character vector
Values: double scalar
Default: `'1'`

Pilot time delay(s) — Pilot time delay

`0.1` (default) | scalar

Total pilot time delay, specified as a double scalar, in seconds. This value typically ranges from 0.1 s to 0.2 s.

Programmatic Use
Block Parameter: `time_delay`
Type: character vector
Values: double scalar
Default: `'0.1'`

Pilot lead constant — Pilot lead constant

`5` (default) | scalar

Pilot lead constant, specified as a double scalar.

Programmatic Use
Block Parameter: `T`
Type: character vector
Values: double scalar
Default: `'5'`

Algorithms

This pilot model is a single input, single output (SISO) model that represents human behavior, based on the transfer function:

$$\frac{u(s)}{e(s)} = \frac{K_p(1 + Ts)}{s}e^{-\tau s}.$$

In this equation:

Variable	Description
K_p	Pilot gain.
T	Lead constant.
τ	Transport delay time caused by the pilot neuromuscular system.
u(s)	Input to the aircraft model and output to the pilot model.
e(s)	Error between the desired pilot value and the actual value.

Version History

Introduced in R2012b

References

[1] Tustin, A., *The Nature of the Operator's Response in Manual Control, and its Implications for Controller Design.* Convention on Automatic Regulators and Servo Mechanisms. May, 1947.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Crossover Pilot Model | Precision Pilot Model | Transport Delay | `linmod`

Simulation 3D Ultrasonic Sensor

Ultrasonic sensor model in 3D simulation environment

Libraries:
Automated Driving Toolbox / Simulation 3D
Aerospace Blockset / Animation / Simulation 3D
UAV Toolbox / Simulation 3D
Simulink 3D Animation / Simulation 3D / Sensors

Description

Note Simulating models with the Simulation 3D Ultrasonic Sensor block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Ultrasonic Sensor to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

The Simulation 3D Ultrasonic Sensor block generates detections from range measurements taken by an ultrasonic sensor mounted on an ego vehicle in a 3D simulation environment rendered using the Unreal Engine from Epic Games. The block calculates range measurements based on the distance between the sensor and the closest point on the detected object.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

Tip The Simulation 3D Scene Configuration block must execute before the Simulation 3D Ultrasonic Sensor block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Ultrasonic Sensor block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Ultrasonic Sensor — `1`

For more information about execution order, see “How 3D Simulation for Aerospace Blockset Works” on page 2-40.

Ports

Output

Has object — Detectable object present in sensor field-of-view
scalar

Detectable object present in the sensor field-of-view, returned as a Boolean scalar. An object is considered detectable if its closest distance to the sensor is greater than the minimum detection-only range specified in the **Detection ranges (m)** parameter.

Has range — Range measurement possible for object present in sensor field-of-view
scalar

Range measurement is possible for an object present in the sensor field-of-view, returned as a Boolean scalar. For any object in the field-of-view, range measurement is possible if its closest distance to the sensor is greater than the minimum-distance range specified in the **Detection ranges (m)** parameter.

Range — Distance measurement to closest object
scalar

Distance measurement to the closest object, returned as a nonnegative scalar, in meters.

Translation — Sensor location
real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Translation** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Rotation — Sensor orientation
real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Rotation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

Sensor identifier — Unique sensor identifier

`1` (default) | positive integer

Specify the unique identifier of the sensor. In a multisensor system, the sensor identifier enables you to distinguish between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$, where N is the highest **Sensor identifier** value among the existing sensor blocks in the model.

Example: 2

Parent name — Name of parent vehicle

`Origin` (default) | vehicle name

Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the simulation 3D vehicle blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

Example: `SimulinkVehicle1`

Mounting location — Sensor mounting location

`Origin` (default)

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle, the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from above, the yaw angle (the orientation angle) is counterclockwise-positive because you are looking in the negative direction of the axis.

Specify offset — Specify offset from mounting location

`off` (default)

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

Relative translation [X, Y, Z] (m) — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “About Aerospace Coordinate Systems” on page 2-8.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

Relative rotation [Roll, Pitch, Yaw] (deg) — Rotational offset relative to mounting location

real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*]. Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “About Aerospace Coordinate Systems” on page 2-8.

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

Sample time — Sample time

`-1` (default) | positive scalar

Sample time of the block, in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Sensor Parameters

Detection ranges (m) — Detection range vector of ultrasonic sensor (m)

`[0.03 0.15 5.5]` (default) | 1-by-3 nonnegative real-valued vector of form `[minDetOnlyRange minDistRange maxDistRange]`

Detection range vector of the ultrasonic sensor, specified as a 1-by-3 nonnegative real-valued vector of the form `[minDetOnlyRange minDistRange maxDistRange]`, where `minDetOnlyRange < minDistRange < maxDistRange`. Units are in meters. These values determine the detections and distance values returned by the ultrasonic sensor.

- When the detected object is at a distance between `minDistRange` and `maxDistRange`, the sensor returns a positive distance value.
- When the detected object is at a distance between `minDetOnlyRange` and `minDistRange`, the sensor detects the object, but cannot determine the distance and returns a value of `0`.
- When the object is at a distance below `minDetOnlyRange` or above `maxDistRange`, the sensor returns an empty cell array.

Horizontal field of view (deg) — Horizontal field of view of ultrasonic sensor

`70` (default) | positive real scalar

Horizontal field of view of ultrasonic sensor, specified as a positive real scalar. This field of view defines the total angular extent spanned by the sensor in the horizontal direction. You must specify the horizontal field of view `horizontalFOV` in the range (0, 180). Units are in degrees.

Vertical field of view (deg) — Vertical field of view of ultrasonic sensor

`35` (default) | positive real scalar

Vertical field of view of ultrasonic sensor, specified as a positive real scalar. This field of view defines the total angular extent spanned by the sensor in the vertical direction. You must specify the vertical field of view in the range (0, 180). Units are in degrees.

Ground Truth

Output location (m) and orientation (rad) — Output location and orientation of sensor

`off` (default) | `on`

Select this parameter to output the translation and rotation of the sensor at the **Translation** and **Rotation** ports, respectively.

Version History

Introduced in R2023b

R2023b: Addition of ground truth parameter

Simulation 3D Ultrasonic Sensor block now has ground truth parameter that you can enable to output the location and orientation of the sensor

R2024a: Requires Simulink 3D Animation

Behavior change in future release

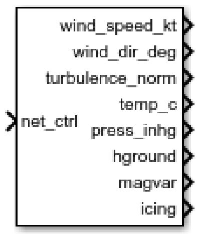
Simulating models with the Simulation 3D Ultrasonic Sensor block requires Simulink 3D Animation. If you had an Aerospace Blockset license prior to R2024a, you might be eligible to continue using Simulation 3D Ultrasonic Sensor to simulate models in the 3D environment. For more information, see Opt-In Offer for Aerospace Blockset Unreal Engine Visualization Users.

See Also

Simulation 3D Scene Configuration

Unpack net_ctrl Packet from FlightGear

Unpack `net_ctrl` variable packet received from FlightGear



Libraries:
Aerospace Blockset / Animation / Flight Simulator Interfaces

Description

The Unpack net_ctrl Packet from FlightGear block unpacks `net_ctrl` variable packets received from FlightGear via the Receive net_ctrl Packet from FlightGear block, and makes them available for the Simulink environment.

The Aerospace Blockset product supports FlightGear versions starting from v2.6. If you are using a FlightGear version older than 2.6, the model displays a notification from the Simulink Upgrade Advisor. Consider using the Upgrade Advisor to upgrade your FlightGear version. For more information, see “Supported FlightGear Versions” on page 2-20.

Ports

Input

net_ctrl — FlightGear packet to be unpacked
array

FlightGear packet to be unpacked, specified as an array.

Data Types: `uint8`

Output

Environment Outputs

wind_speed_kt — Wind speed
scalar

Wind speed, specified as a scalar, in knots.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

wind_dir_deg — Wind direction
scalar

Wind direction, specified as a scalar, in deg.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

turbulence_norm — Turbulence norm
scalar

Turbulence norm, specified as a scalar.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

temp_c — Ambient temperature
scalar

Ambient temperature, specified as a scalar, in deg C.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

press_inhg — Ambient pressure
scalar

Ambient pressure, specified as a scalar, in inHg.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

hground — Ground elevation
scalar

Ground elevation, specified as a scalar, in m.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

magvar — Local magnetic variation
scalar

Local magnetic variation, specified as a scalar.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `double`

icing — Icing status
scalar

Icing status, specified as a scalar, in deg.

Dependencies

To enable this port, select the **Show environment outputs** check box.

Data Types: `uint32`

Control Surface Position Inputs

aileron — Normalized aileron position
1 | scalar

Normalized aileron position [-1,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

elevator — Normalized elevator position
1 | scalar

Normalized elevator position [-1,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

rudder — Normalized rudder position
1 | scalar

Normalized rudder position [-1,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

aileron_trim — Normalized aileron trim position
scalar

Normalized aileron trim position [-1,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

elevator_trim — Normalized elevator trim position
1 | scalar

Normalized elevator trim position [-1,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

rudder_trim — Normalized rudder trim position
1 | scalar

Normalized rudder trim position [-1,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

flaps — Normalized flaps position
1 | scalar

Normalized flaps position [-0,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `double`

spoilers — Normalized spoilers position
1 | scalar

Normalized spoilers position [0,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `single`

speedbrake — Normalized speedbrake position
1 | scalar

Normalized speedbrake position [0,1], specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `single`

flaps_power — Power for flaps
1 | scalar

Power for flaps, specified as a scalar. A value of `1` indicates that power is available.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `uint32`

flap_motor_ok — Flap motor powered
scalar

Flap motor powered, specified as a scalar.

Dependencies

To enable this port, select the **Show control surface position outputs** check box.

Data Types: `uint32`

Engine/Fuel Outputs

num_engines — Number of valid engines
scalar

Number of valid engines, specified as a scalar.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

master_bat — Master battery switch
vector

Master battery switch, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

master_alt — Master alternator switch
vector

Master alternator switch, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

magnetos — Magnetos switch
scalar

Magnetos switch, specified as a scalar.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

starter_power — Power to start motor
1 | vector

Power to starter motor, specified as a vector. A value of 1 indicates that power is available.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

throttle — Normalized throttle position
1 | vector

Normalized throttle position [0,1], specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `double`

mixture — Normalized mixture lever position
1 | vector

Normalized mixture lever position [0,1], specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `double`

condition — Normalized condition
1 | vector

Normalized condition [0,1], specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

fuel_pump_power — Normalized speedbrake position
1 | scalar

Power to fuel pump, specified as a vector. A value of 1 indicates that pump is on.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

prop_advance — Propeller advance
1 | vector

Propeller advance [0,1], specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `double`

feed_tank_to — Feed tank to switch
vector

Feed tank to switch, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

reverse — Reverse switch
vector

Reverse switch, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

engine_ok — Engine status indicator
vector

Engine status indicator, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

mag_left_ok — Left magneto status indicator
vector

Left magneto status indicator, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

mag_right_ok — Right magneto status indicator
vector

Right magneto status indicator, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

spark_plugs_ok — Normalized speedbrake position
vector

Spark plugs status indicator, specified as a vector. A value of `0` indicates that the plugs have failed.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

oil_press_status — Oil pressure status indicator
`0` | `1` | `2` | scalar

Oil pressure status indicator, specified as a vector.

- `0` — Normal oil pressure
- `1` — Low oil pressure
- `2` — Failed oil pressure

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

fuel_pump_ok — Fuel management status indicator
vector

Fuel management status indicator, specified as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

num_tanks — Number of valid tanks
scalar

Number of valid tanks, specified as a scalar.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

fuel_selector — Fuel selector
scalar

Fuel selector, specified as a vector.

- `0` — Off
- `1` — On

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `single`

xfer_pump — Specify transfer
vector

Specifies transfer from array value to tank, specified by value as a vector.

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `uint32`

cross_feed — Cross feed valve
scalar

Cross feed valve, specified as a scalar.

- `0` — False
- `1` — On

Dependencies

To enable this port, select the **Show engine/fuel outputs** check box.

Data Types: `single`

Landing Gear Outputs

brake_left — Left brake pedal position pilot
scalar

Left brake pedal position pilot, specified as a scalar.

Dependencies

To enable this port, select the **Show landing gear outputs** check box.

Data Types: `double`

brake_right — Right brake pedal position pilot
scalar

Right brake pedal position pilot, specified as a scalar.

Dependencies

To enable this port, select the **Show landing gear outputs** check box.

Data Types: `double`

copilot_brake_left — Left brake pedal position pilot
scalar

Left brake pedal position pilot, specified as a scalar.

Dependencies

To enable this port, select the **Show landing gear outputs** check box.

Data Types: `double`

copilot_brake_right — Right brake pedal position pilot
scalar

Right brake pedal position pilot, specified as a scalar.

Dependencies

To enable this port, select the **Show landing gear outputs** check box.

Data Types: `double`

brake_parking — Brake parking position
scalar

Brake parking position, specified as a scalar.

Dependencies

To enable this port, select the **Show landing gear outputs** check box.

Data Types: `double`

gear_handle — Gear handle position
scalar

Gear handle position, specified as a scalar.

- `0` — Gear handle up
- `1` — Gear handle down

Dependencies

To enable this port, select the **Show landing gear outputs** check box.

Data Types: `uint32`

Avionic Outputs

master_avionics — Master avionics switch
scalar

Master avionics switch, specified as a scalar.

Dependencies

To enable this port, select the **Show avionic outputs** check box.

Data Types: `uint32`

comm_1 — Comm 1 frequency
scalar

Comm 1 frequency, specified as a scalar, in Hz.

Dependencies

To enable this port, select the **Show avionic outputs** check box.

Data Types: `double`

comm_2 — Comm 2 frequency
scalar

Comm 2 frequency, specified as a scalar, in Hz.

Dependencies

To enable this port, select the **Show avionic outputs** check box.

Data Types: `double`

nav_1 — Nav 1 frequency
scalar

Nav 1 frequency, specified as a scalar, in Hz.

Dependencies

To enable this port, select the **Show avionic outputs** check box.

Data Types: `double`

nav_2 — Nav 2 frequency
scalar

Nav 2 frequency, specified as a scalar, in Hz.

Dependencies

To enable this port, select the **Show avionic outputs** check box.

Data Types: `double`

Parameters

Show control surface position outputs — Control surface position outputs

`off` (default) | `on`

Select this check box to include the control surface position outputs from the FlightGear `net_ctrl` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 1: Control surface position outputs

Name	Units	Type	Width	Description
aileron	1 (dimensionless)	double	1	Normalized aileron position [-1,1]
elevator	1 (dimensionless)	double	1	Normalized elevator position [-1,1]
rudder	1 (dimensionless)	double	1	Normalized rudder position [-1,1]
aileron_trim	1 (dimensionless)	double	1	Normalized aileron trim position [-1,1]
elevator_trim	1 (dimensionless)	double	1	Normalized elevator trim position [-1,1]
rudder_trim	1 (dimensionless)	double	1	Normalized rudder trim position [-1,1]
flaps	1 (dimensionless)	double	1	Normalized flaps position [-0,1]
spoilers	1 (dimensionless)	double	1	Normalized spoilers position [0,1]
speedbrake	1 (dimensionless)	double	1	Normalized speedbrake position [0,1]
flaps_power	1 (dimensionless)	uint32	1	Power for flaps (`1` = power available)
flap_motor_ok	—	uint32	1	Flap motor powered

Programmatic Use
Block Parameter: `ShowControlSurfacePositionOutputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Show engine/fuel outputs — Engine/fuel outputs

`off` (default) | `on`

Select this check box to include the engine and fuel outputs from the FlightGear `net_ctrl` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 2: Engine/fuel outputs

Name	Units	Type	Width	Description
num_engines	—	uint32	1	Number of valid engines
master_bat	—	uint32	4	Master battery switch
master_alt	—	uint32	4	Master alternator switch
magnetos	—	uint32	4	Magnetos switch
starter_power	—	uint32	4	Power to starter motor (1 = starter power available)
throttle	1 (dimensionless)	double	4	Normalized throttle position [0,1]
mixture	1 (dimensionless)	double	4	Normalized mixture lever position [0,1]
condition	1 (dimensionless)	double	4	Normalized condition [0,1]
fuel_pump_power	—	uint32	4	Power to fuel pump 1 = on)
prop_advance	1 (dimensionless)	double	4	Propeller advance [0,1]
feed_tank_to	—	uint32	4	Feed tank to switch
reverse	—	uint32	4	Reverse switch
engine_ok	—	uint32	4	Engine status indicator
mag_left_ok	—	uint32	4	Left magneto status indicator
mag_right_ok	—	uint32	4	Right magneto status indicator
spark_plugs_ok	—	uint32	4	Spark plugs status indicator (0 = failed plugs)
oil_press_status	—	uint32	4	Oil pressure status indicator (0 = normal, 1 = low, 2 = full failure)
fuel_pump_ok	—	uint32	4	Fuel management status indicator
num_tanks	—	uint32	1	Number of valid tanks
fuel_selector	—	uint32	8	Fuel selector. (0 = off, 1 = on)
xfer_pump	—	uint32	5	Specifies transfer from array value to tank specified by value
cross_feed	—	uint32	1	Cross feed valve (0 = false, 1 = on)

Programmatic Use
Block Parameter: `ShowEngineFuelOutputs`
Type: character vector
Values: `'off' | 'on'`

Default: 'off'

Show landing gear outputs — Landing gear outputs

off (default) | on

Select this check box to include the landing gear outputs from the FlightGear net_ctrl data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 3: Landing gear outputs

Name	Units	Type	Width	Description
brake_left	—	double	1	Left brake pedal position pilot
brake_right	—	double	1	Right brake pedal position pilot
copilot_brake_left	—	double	1	Left brake pedal position copilot
copilot_brake_right	—	double	1	Right brake pedal position copilot
brake_parking	—	double	1	Brake parking position
gear_handle	—	uint32	1	Gear handle position (1 = gear handle down, 0 = gear handle up)

Programmatic Use
Block Parameter: ShowLandingGearOutputs
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Show avionic outputs — Avionic outputs

off (default) | on

Select this check box to include the avionic outputs from the FlightGear net_ctrl data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 4: Avionics outputs

Name	Units	Type	Width	Description
master_avionics	—	uint32	1	Master avionics switch
comm_1	Hz	double	1	Comm 1 frequency
comm_2	Hz	double	1	Comm 2 frequency
nav_1	Hz	double	1	Nav 1 frequency
nav_2	Hz	double	1	Nav 2 frequency

Programmatic Use
Block Parameter: `ShowAvionicOutputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'off'`

Show environment outputs — Environment outputs

on (default) | `off`

Select this check box to include the environment outputs from the FlightGear `net_ctrl` data packet.

Dependencies

Select this check box to enable these input ports.

Signal Group 5: Environment outputs

Name	Units	Type	Width	Description
wind_speed_kt	knot	double	1	Wind speed
wind_dir_deg	deg	double	1	Wind direction
turbulence_norm	—	double	1	Turbulence norm
temp_c	deg C	double	1	Ambient temperature
press_inhg	inHg	double	1	Ambient pressure
hground	m	double	1	Ground elevation
magvar	deg	double	1	Local magnetic variation
icing	-	uint32	1	Icing status

Programmatic Use
Block Parameter: `ShowEnvironmentOutputs`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

Sample time — Sample time

1/30 (default) | scalar

Specify the sample time (-1 for inherited), as a scalar.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar
Default: `'1/30'`

Version History

Introduced in R2012a

See Also

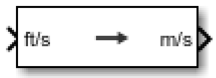
FlightGear Preconfigured 6DoF Animation | Generate Run Script | Pack net_fdm Packet for FlightGear | Receive net_ctrl Packet from FlightGear | Send net_fdm Packet to FlightGear

Topics

"Flight Simulator Interface" on page 2-20
"Work with the Flight Simulator Interface" on page 2-24

Velocity Conversion

Convert from velocity units to desired velocity units



Libraries:
Aerospace Blockset / Utilities / Unit Conversions

Description

The Velocity Conversion block computes the conversion factor from specified input velocity units to specified output velocity units and applies the conversion factor to the input signal.

The Velocity Conversion block port labels change based on the input and output units selected from the **Initial unit** and the **Final unit** lists.

Ports

Input

Port_1 — Velocity
scalar | array

Velocity, specified as a scalar or array, in initial velocity units.

Dependencies

The input port label depends on the **Initial unit** setting.

Data Types: `double`

Output

Port_1 — Velocity
scalar | array

Velocity, returned as a scalar or array, in final velocity units.

Dependencies

The output port label depends on the **Final unit** setting.

Data Types: `double`

Parameters

Initial unit — Input units

`ft/s` (default) | `m/s` | `km/s` | `in/s` | `km/h` | `mph` | `kts` | `ft/min`

Input units, specified as:

`m/s`	Meters per second
`ft/s`	Feet per second
`km/s`	Kilometers per second
`in/s`	Inches per second
`km/h`	Kilometers per hour
`mph`	Miles per hour
`kts`	Nautical miles per hour
`ft/min`	Feet per minute

Dependencies

The input port label depends on the **Initial unit** setting.

Programmatic Use

Block Parameter: `IU`
Type: character vector
Values: `'m/s'` | `'ft/s'` | `'km/s'` | `'in/s'` | `'km/h'` | `'mph'` | `'kts'` | `'ft/min'`
Default: `'ft/s'`

Final unit — Output units

`m/s` (default) | `ft/s` | `km/s` | `in/s` | `km/h` | `mph` | `kts` | `ft/min`

Output units, specified as:

`m/s`	Meters per second
`ft/s`	Feet per second
`km/s`	Kilometers per second
`in/s`	Inches per second
`km/h`	Kilometers per hour
`mph`	Miles per hour
`kts`	Nautical miles per hour
`ft/min`	Feet per minute

Dependencies

The output port label depends on the **Final unit** setting.

Programmatic Use

Block Parameter: `OU`
Type: character vector
Values: `'m/s'` | `'ft/s'` | `'km/s'` | `'in/s'` | `'km/h'` | `'mph'` | `'kts'` | `'ft/min'`
Default: `'m/s'`

Version History

Introduced before R2006a

Extended Capabilities

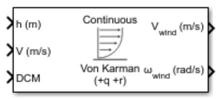
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Acceleration Conversion | Angle Conversion | Angular Acceleration Conversion | Angular Velocity Conversion | Density Conversion | Force Conversion | Length Conversion | Mass Conversion | Pressure Conversion | Temperature Conversion

Von Karman Wind Turbulence Model (Continuous)

Generate continuous wind turbulence with Von Kármán velocity spectra



Libraries:
Aerospace Blockset / Environment / Wind

Description

The Von Kármán Wind Turbulence Model (Continuous) block uses the Von Kármán spectral representation to add turbulence to the aerospace model by passing band-limited white noise through appropriate forming filters. This block implements the mathematical representation in the Military Specification MIL-F-8785C, Military Handbook MIL-HDBK-1797, and Military Handbook MIL-HDBK-1797B. For more information, see "Algorithms" on page 5-1279.

Limitations

- The frozen turbulence field assumption is valid for the cases of mean-wind velocity.
- The root-mean-square turbulence velocity, or intensity, is small relative to the aircraft ground speed.
- The turbulence model describes an average of all conditions for clear air turbulence because the following factors are not incorporated into the model:
  - Terrain roughness
  - Lapse rate
  - Wind shears
  - Mean wind magnitude
  - Other meteorological factions (except altitude)

Ports

Input

h — Altitude
scalar

Altitude, specified as a scalar, in selected units.

Data Types: `double`

V — Aircraft speed
scalar

Aircraft speed, specified as a scalar, in selected units.

Data Types: `double`

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix representing the flat Earth coordinates to body-fixed axis coordinates.

Data Types: `double`

Output

V_{wind} — Turbulence velocities
three-element vector

Turbulence velocities, returned as a three-element vector in the same body coordinate reference as the **DCM** input, in specified units.

Data Types: `double`

ω_{wind} — Turbulence angular rates
three-element vector

Turbulence angular rates, specified as a three-element vector, in radians per second.

Data Types: `double`

Parameters

Units — Wind speed units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Units of wind speed due to turbulence, specified as:

Units	Wind Velocity	Altitude	Air Speed
`Metric (MKS)`	Meters/second	Meters	Meters/second
`English (Velocity in ft/s)`	Feet/second	Feet	Feet/second
`English (Velocity in kts)`	Knots	Feet	Knots

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Specification — Military reference

`MIL-F-8785C` (default) | `MIL-HDBK-1797` | `MIL-HDBK-1797B`

Military reference, which affects the application of turbulence scale lengths in the lateral and vertical directions, specified as `MIL-F-8785C`, `MIL-HDBK-1797`, or `MIL-HDBK-1797B`.

Programmatic Use
Block Parameter: `spec`
Type: character vector
Values: `'MIL-F-8785C'` | `'MIL-HDBK-1797'` | `'MIL-HDBK-1797B'`
Default: `'MIL-F-8785C'`

Model type — Turbulence model

`Continuous Von Karman (+q -r)` (default) | `Continuous Von Karman (+q +r)` | `Continuous Von Karman (-q +r)` | `Continuous Dryden (+q -r)` | `Continuous Dryden (+q +r)` | `Continuous Dryden (-q +r)` | `Discrete Dryden (+q -r)` | `Discrete Dryden (+q +r)` | `Discrete Dryden (-q +r)`

Wind turbulence model, specified as:

`Continuous Von Karman (+q -r)`	Use continuous representation of Von Kármán velocity spectra with positive vertical and negative lateral angular rates spectra.
`Continuous Von Karman (+q +r)`	Use continuous representation of Von Kármán velocity spectra with positive vertical and lateral angular rates spectra.
`Continuous Von Karman (-q +r)`	Use continuous representation of Von Kármán velocity spectra with negative vertical and positive lateral angular rates spectra.
`Continuous Dryden (+q -r)`	Use continuous representation of Dryden velocity spectra with positive vertical and negative lateral angular rates spectra.
`Continuous Dryden (+q +r)`	Use continuous representation of Dryden velocity spectra with positive vertical and lateral angular rates spectra.
`Continuous Dryden (-q +r)`	Use continuous representation of Dryden velocity spectra with negative vertical and positive lateral angular rates spectra.
`Discrete Dryden (+q -r)`	Use discrete representation of Dryden velocity spectra with positive vertical and negative lateral angular rates spectra.
`Discrete Dryden (+q +r)`	Use discrete representation of Dryden velocity spectra with positive vertical and lateral angular rates spectra.
`Discrete Dryden (-q +r)`	Use discrete representation of Dryden velocity spectra with negative vertical and positive lateral angular rates spectra.

The Continuous Von Kármán selections conform to the transfer function descriptions.

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'Continuous Von Karman (+q +r)'` | `'Continuous Von Karman (-q +r)'` | `'Continuous Dryden (+q -r)'` | `'Continuous Dryden (+q +r)'` | `'Continuous Dryden`

`(-q +r)'` | `'Discrete Dryden (+q -r)'` | `'Discrete Dryden (+q +r)'` | `'Discrete Dryden (-q +r)'`
Default: `'Continuous Von Karman (+q +r)'`

Wind speed at 6 m defines the low altitude intensity — Measured wind speed

`15` (default) | real scalar

Measured wind speed at a height of 20 feet (6 meters), specified as a real scalar, which provides the intensity for the low-altitude turbulence model.

Programmatic Use
Block Parameter: `W20`
Type: character vector
Values: real scalar
Default: `'15'`

Wind direction at 6 m (degrees clockwise from north) — Measured wind direction

`0` (default) | real scalar

Measured wind direction at a height of 20 feet (6 meters), specified as a real scalar, which is an angle to aid in transforming the low-altitude turbulence model into a body coordinates.

Programmatic Use
Block Parameter: `Wdeg`
Type: character vector
Values: real scalar
Default: `'0'`

Probability of exceedance of high-altitude intensity — Turbulence intensity

`10^-2 - Light` (default) | `10^-1` | `2x10^-1` | `10^-3 - Moderate` | `10^-4` | `10^-5 - Severe` | `10^-6`

Probability of the turbulence intensity being exceeded, specified as `10^-2 - Light`, `10^-1`, `2x10^-1`, `10^-3 - Moderate`, `10^-4`, `10^-5 - Severe`, or `10^-6`. Above 2000 feet, the turbulence intensity is determined from a lookup table that gives the turbulence intensity as a function of altitude and the probability of the turbulence intensity being exceeded.

Programmatic Use
Block Parameter: `TurbProb`
Type: character vector
Values: `'2x10^-1'` | `'10^-1'` | `'10^-2 - Light'` | `'10^-3 - Moderate'` | `'10^-4'` | `'10^-5 - Severe'` | `'10^-6'`
Default: `'10^-2 - Light'`

Scale length at medium/high altitudes — Turbulence scale length

`762` (default) | real scalar

Turbulence scale length above 2000 feet, specified as a real scalar. This length is assumed constant.

From the military specifications, 1750 feet is recommended for the longitudinal turbulence scale length of the Dryden spectra.

Note An alternate scale length value changes the power spectral density asymptote and gust load.

Programmatic Use
Block Parameter: `L_high`
Type: character vector
Values: real scalar
Default: `'762'`

Wingspan — Wingspan

`10` (default) | real scalar

Wingspan, specified as a real scalar, which is required in the calculation of the turbulence on the angular rates.

Programmatic Use
Block Parameter: `Wingspan`
Type: character vector
Values: real scalar
Default: `'10'`

Band limited noise sample time (seconds) — Noise sample time

`0.1` (default) | real scalar

Noise sample time, specified as a real scalar, at which the unit variance white noise signal is generated.

Programmatic Use
Block Parameter: `ts`
Type: character vector
Values: real scalar
Default: `'0.1'`

Random noise seeds — Noise seeds [ug vg wg pg]

`[23341 23342 23343 23344]` (default) | four-element vector

Random noise seeds, specified as a four-element vector, which are used to generate the turbulence signals, one for each of the three velocity components and one for the roll rate:

The turbulences on the pitch and yaw angular rates are based on further shaping of the outputs from the shaping filters for the vertical and lateral velocities.

Programmatic Use
Block Parameter: `Seed`
Type: character vector
Values: four-element vector
Default: `'[23341 23342 23343 23344]'`

Turbulence on — Turbulence signals

`on` (default) | `off`

To generate the turbulence signals, select this check box.

Programmatic Use
Block Parameter: `T_on`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Algorithms

According to the military references, turbulence is a stochastic process defined by velocity spectra. For an aircraft flying at a speed V through a frozen turbulence field with a spatial frequency of Ω radians per meter, the circular frequency ω is calculated by multiplying V by Ω. The following table displays the component spectra functions:

	MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
Longitudinal		
$\Phi_u(\omega)$	$\frac{2\sigma_u^2 L_u}{\pi V} \cdot \frac{1}{\left[1 + \left(1.339 L_u \frac{\omega}{V}\right)^2\right]^{5/6}}$	$\frac{2\sigma_u^2 L_u}{\pi V} \cdot \frac{1}{\left[1 + \left(1.339 L_u \frac{\omega}{V}\right)^2\right]^{5/6}}$
$\Phi_p(\omega)$	$\frac{\sigma_w^2}{V L_w} \cdot \frac{0.8\left(\frac{\pi L_w}{4b}\right)^{1/3}}{1 + \left(\frac{4b\omega}{\pi V}\right)^2}$	$\frac{\sigma_w^2}{2V L_w} \cdot \frac{0.8\left(\frac{2\pi L_w}{4b}\right)^{1/3}}{1 + \left(\frac{4b\omega}{\pi V}\right)^2}$
Lateral		
$\Phi_v(\omega)$	$\frac{\sigma_v^2 L_v}{\pi V} \cdot \frac{1 + \frac{8}{3}\left(1.339 L_v \frac{\omega}{V}\right)^2}{\left[1 + \left(1.339 L_v \frac{\omega}{V}\right)^2\right]^{11/6}}$	$\frac{2\sigma_v^2 L_v}{\pi V} \cdot \frac{1 + \frac{8}{3}\left(2.678 L_v \frac{\omega}{V}\right)^2}{\left[1 + \left(2.678 L_v \frac{\omega}{V}\right)^2\right]^{11/6}}$
$\Phi_r(\omega)$	$\frac{\mp\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{3b\omega}{\pi V}\right)^2} \cdot \Phi_v(\omega)$	$\frac{\mp\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{3b\omega}{\pi V}\right)^2} \cdot \Phi_v(\omega)$
Vertical		
$\Phi_w(\omega)$	$\frac{\sigma_w^2 L_w}{\pi V} \cdot \frac{1 + \frac{8}{3}\left(1.339 L_w \frac{\omega}{V}\right)^2}{\left[1 + \left(1.339 L_w \frac{\omega}{V}\right)^2\right]^{11/6}}$	$\frac{2\sigma_w^2 L_w}{\pi V} \cdot \frac{1 + \frac{8}{3}\left(2.678 L_w \frac{\omega}{V}\right)^2}{\left[1 + \left(2.678 L_w \frac{\omega}{V}\right)^2\right]^{11/6}}$
$\Phi_q(\omega)$	$\frac{\pm\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{4b\omega}{\pi V}\right)^2} \cdot \Phi_w(\omega)$	$\frac{\pm\left(\frac{\omega}{V}\right)^2}{1 + \left(\frac{4b\omega}{\pi V}\right)^2} \cdot \Phi_w(\omega)$

The variable b represents the aircraft wingspan. The variables L_u, L_v, L_w represent the turbulence scale lengths. The variables σ_u, σ_v, σ_w represent the turbulence intensities:

The spectral density definitions of turbulence angular rates are defined in the references as three variations, which are displayed in the following table:

$$p_g = \frac{\partial w_g}{\partial y} \qquad q_g = \frac{\partial w_g}{\partial x} \qquad r_g = -\frac{\partial v_g}{\partial x}$$

$$p_g = \frac{\partial w_g}{\partial y} \qquad q_g = \frac{\partial w_g}{\partial x} \qquad r_g = \frac{\partial v_g}{\partial x}$$

$$p_g = -\frac{\partial w_g}{\partial y} \qquad q_g = -\frac{\partial w_g}{\partial x} \qquad r_g = \frac{\partial v_g}{\partial x}$$

The variations affect only the vertical (q_g) and lateral (r_g) turbulence angular rates.

Keep in mind that the longitudinal turbulence angular rate spectrum, $\Phi_p(\omega)$, is a rational function. The rational function is derived from curve-fitting a complex algebraic function, not the vertical turbulence velocity spectrum, $\Phi_w(\omega)$, multiplied by a scale factor. Because the turbulence angular rate spectra contribute less to the aircraft gust response than the turbulence velocity spectra, it may explain the variations in their definitions.

The variations lead to the following combinations of vertical and lateral turbulence angular rate spectra.

Vertical	Lateral
$\Phi_q(\omega)$	$-\Phi_r(\omega)$
$\Phi_q(\omega)$	$\Phi_r(\omega)$
$-\Phi_q(\omega)$	$\Phi_r(\omega)$

To generate a signal with the correct characteristics, a unit variance, band-limited white noise signal is passed through forming filters. The forming filters are approximations of the Von Kármán velocity spectra which are valid in a range of normalized frequencies of less than 50 radians. These filters can be found in both the Military Handbook MIL-HDBK-1797 and the reference by Ly and Chan.

The following two tables display the transfer functions.

	MIL-F-8785C
Longitudinal	
$H_u(s)$	$\dfrac{\sigma_u\sqrt{\frac{2}{\pi}\cdot\frac{L_u}{V}}\left(1 + 0.25\frac{L_u}{V}s\right)}{1 + 1.357\frac{L_u}{V}s + 0.1987\left(\frac{L_u}{V}\right)^2 s^2}$
$H_p(s)$	$\sigma_w\sqrt{\dfrac{0.8}{V}}\cdot\dfrac{\left(\frac{\pi}{4b}\right)^{1/6}}{L_w{}^{1/3}\left(1 + \left(\frac{4b}{\pi V}\right)s\right)}$
Lateral	
$H_v(s)$	$\dfrac{\sigma_v\sqrt{\frac{1}{\pi}\cdot\frac{L_v}{V}}\left(1 + 2.7478\frac{L_v}{V}s + 0.3398\left(\frac{L_v}{V}\right)^2 s^2\right)}{1 + 2.9958\frac{L_v}{V}s + 1.9754\left(\frac{L_v}{V}\right)^2 s^2 + 0.1539\left(\frac{L_v}{V}\right)^3 s^3}$

	MIL-F-8785C
$H_r(s)$	$\dfrac{\mp\frac{s}{V}}{\left(1+\left(\frac{3b}{\pi V}\right)s\right)} \cdot H_v(s)$
Vertical	
$H_w(s)$	$\dfrac{\sigma_w\sqrt{\frac{1}{\pi}\cdot\frac{L_w}{V}}\left(1+2.7478\frac{L_w}{V}s+0.3398\left(\frac{L_w}{V}\right)^2 s^2\right)}{1+2.9958\frac{L_w}{V}s+1.9754\left(\frac{L_w}{V}\right)^2 s^2+0.1539\left(\frac{L_w}{V}\right)^3 s^3}$
$H_q(s)$	$\dfrac{\pm\frac{s}{V}}{\left(1+\left(\frac{4b}{\pi V}\right)s\right)} \cdot H_w(s)$

	MIL-HDBK-1797 and MIL-HDBK-1797B
Longitudinal	
$H_u(s)$	$\dfrac{\sigma_u\sqrt{\frac{2}{\pi}\cdot\frac{L_u}{V}}\left(1+0.25\frac{L_u}{V}s\right)}{1+1.357\frac{L_u}{V}s+0.1987\left(\frac{L_u}{V}\right)^2 s^2}$
$H_p(s)$	$\sigma_w\sqrt{\dfrac{0.8}{V}} \cdot \dfrac{\left(\frac{\pi}{4b}\right)^{1/6}}{(2L_w)^{1/3}\left(1+\left(\frac{4b}{\pi V}\right)s\right)}$
Lateral	
$H_v(s)$	$\dfrac{\sigma_v\sqrt{\frac{1}{\pi}\cdot\frac{2L_v}{V}}\left(1+2.7478\frac{2L_v}{V}s+0.3398\left(\frac{2L_v}{V}\right)^2 s^2\right)}{1+2.9958\frac{2L_v}{V}s+1.9754\left(\frac{2L_v}{V}\right)^2 s^2+0.1539\left(\frac{2L_v}{V}\right)^3 s^3}$
$H_r(s)$	$\dfrac{\mp\frac{s}{V}}{\left(1+\left(\frac{3b}{\pi V}\right)s\right)} \cdot H_v(s)$
Vertical	
$H_w(s)$	$\dfrac{\sigma_w\sqrt{\frac{1}{\pi}\cdot\frac{2L_w}{V}}\left(1+2.7478\frac{2L_w}{V}s+0.3398\left(\frac{2L_w}{V}\right)^2 s^2\right)}{1+2.9958\frac{2L_w}{V}s+1.9754\left(\frac{2L_w}{V}\right)^2 s^2+0.1539\left(\frac{2L_w}{V}\right)^3 s^3}$
$H_q(s)$	$\dfrac{\pm\frac{s}{V}}{\left(1+\left(\frac{4b}{\pi V}\right)s\right)} \cdot H_w(s)$

Divided into two distinct regions, the turbulence scale lengths and intensities are functions of altitude.

Note The same transfer functions result after evaluating the turbulence scale lengths. The differences in turbulence scale lengths and turbulence transfer functions balance offset.

Low-Altitude Model (Altitude < 1000 feet)

According to the military references, the turbulence scale lengths at low altitudes, where h is the altitude in feet, are represented in the following table:

MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$L_w = h$ $L_u = L_v = \frac{h}{(0.177 + 0.000823h)^{1.2}}$	$2L_w = h$ $L_u = 2L_v = \frac{h}{(0.177 + 0.000823h)^{1.2}}$

The turbulence intensities are given below, where W_{20} is the wind speed at 20 feet (6 m). Typically for light turbulence, the wind speed at 20 feet is 15 knots; for moderate turbulence, the wind speed is 30 knots; and for severe turbulence, the wind speed is 45 knots.

$$\sigma_w = 0.1W_{20}$$

$$\frac{\sigma_u}{\sigma_w} = \frac{\sigma_v}{\sigma_w} = \frac{1}{(0.177 + 0.000823h)^{0.4}}$$

The turbulence axes orientation in this region is defined as follows:

- Longitudinal turbulence velocity, u_g, aligned along the horizontal relative mean wind vector.
- Vertical turbulence velocity, w_g, aligned with vertical relative mean wind vector.

At this altitude range, the output of the block is transformed into body coordinates.

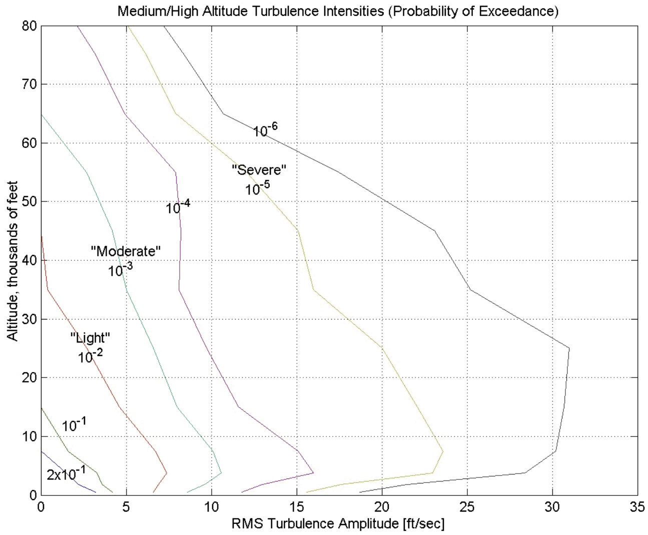
Medium/High Altitudes (Altitude > 2000 feet)

For medium to high altitudes the turbulence scale lengths and intensities are based on the assumption that the turbulence is isotropic. In the military references, the scale lengths are represented by the following equations:

MIL-F-8785C	MIL-HDBK-1797 and MIL-HDBK-1797B
$L_u = L_v = L_w = 2500$ ft	$L_u = 2L_v = 2L_w = 2500$ ft

The turbulence intensities are determined from a lookup table that provides the turbulence intensity as a function of altitude and the probability of the turbulence intensity being exceeded. The relationship of the turbulence intensities is represented in the following equation: $\sigma_u=\sigma_v=\sigma_w$.

The turbulence axes orientation in this region is defined as being aligned with the body coordinates:



Between Low and Medium/High Altitudes (1000 feet < Altitude < 2000 feet)

At altitudes between 1000 feet and 2000 feet, the turbulence velocities and turbulence angular rates are determined by linearly interpolating between the value from the low altitude model at 1000 feet transformed from mean horizontal wind coordinates to body coordinates and the value from the high altitude model at 2000 feet in body coordinates.

Version History

Introduced in R2006b

References

[1] U.S. Military Handbook MIL-HDBK-1797B, April 9, 2012.

[2] U.S. Military Handbook MIL-HDBK-1797, December 19, 1997.

[3] U.S. Military Specification MIL-F-8785C, November 5, 1980.

[4] Chalk, Charles, T.P. Neal, T.M. Harris, Francis E. Pritchard, and Robert J. Woodcock. "Background Information and User Guide for MIL-F-8785B(ASG), `Military Specification-Flying Qualities of Piloted Airplanes'," AD869856. Buffalo, NY: Cornell Aeronautical Laboratory, August 1969.

[5] Hoblit, Frederic M., *Gust Loads on Aircraft: Concepts and Applications*. AIAA Education Series, 1988.

[6] Ly, U. and Y. Chan. "Time-Domain Computation of Aircraft Gust Covariance Matrices." AIAA Paper 80-1615. Presented at the Atmospheric Flight Mechanics Conference, Danvers, MA, August 11-13, 1980.

[7] McRuer, Duane, Irving Ashkenas, and Dunstan Graham. *Aircraft Dynamics and Automatic Control*. Princeton: Princeton University Press, July 1990.

[8] Moorhouse, David J. and Robert J. Woodcock. "Background Information and User Guide for MIL-F-8785C, 'Military Specification-Flying Qualities of Piloted Airplanes'." ADA119421. Flight Dynamic Laboratory, July 1982.

[9] McFarland, R. "A Standard Kinematic Model for Flight Simulation at NASA-Ames." NASA CR-2497. Computer Sciences Corporation, January 1975.

[10] Tatom, Frank B., Stephen R. Smith, and George H. Fichtl. "Simulation of Atmospheric Turbulent Gusts and Gust Gradients," AIAA Paper 81-0300. Aerospace Sciences Meeting, St. Louis, MO., January 12-15, 1981.

[11] Yeager, Jessie. "Implementation and Testing of Turbulence Models for the F18-HARV Simulation." NASA CR-1998-206937. Hampton, VA: Lockheed Martin Engineering & Sciences, March 1998.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Dryden Wind Turbulence Model (Continuous) | Dryden Wind Turbulence Model (Discrete) | Discrete Wind Gust Model | Wind Shear Model

WGS84 Gravity Model

Implement 1984 World Geodetic System (WGS84) representation of Earth's gravity

Libraries:
Aerospace Blockset / Environment / Gravity

Description

The WGS84 Gravity Model block implements the mathematical representation of the geocentric equipotential ellipsoid of the World Geodetic System (WGS84). The block output is the Earth gravity at a specific location. To control gravity precision, use the **Type of gravity model** parameter.

The block icon displays the input and output units selected from the **Units** list.

Limitations

- The WGS84 gravity calculations are based on the assumption of a geocentric equipotential ellipsoid of revolution. Since the gravity potential is assumed to be the same everywhere on the ellipsoid, there must be a specific theoretical gravity potential that can be uniquely determined from the four independent constants defining the ellipsoid.
- Use of the WGS84 Taylor Series model should be limited to low geodetic heights. It is sufficient near the surface when submicrogal precision is not necessary. At medium and high geodetic heights, it is less accurate.
- The WGS84 Close Approximation model gives results with submicrogal precision.
- To predict and determine a satellite orbit with high accuracy, use the EGM96 through degree and order 70.

Ports

Input

μ l h — Position in geodetic latitude, longitude, and altitude
three-element vector | *M*-by-3 array

Position in geodetic latitude, longitude, and altitude, specified as a three-element vector or *M*-by-3 array, in selected units. Altitude must be less than 20,000 m (approximately 65,620 feet).

Data Types: `double`

JD — Julian date
scalar | value greater than 2451545

Julian date, specified as a scalar. The year must be after January 1, 2000 (2451545).

Dependencies

To enable this port, select **Input Julian date**.

Data Types: `double`

Output

Output 1 — Gravity
three-element vector | *M*-by-3 array

Gravity in the north-east-down (NED) coordinate system.

Gravity Model Method	Output
Taylor Series and Close Approximation	Output only normal gravity (down in the NED coordinate system).
Exact	Both normal and tangent gravity (down and north in the NED coordinate system).

Data Types: `double`

Parameters

Type of gravity model — Gravity model method

`WGS84 Taylor Series` (default) | `WGS84 Close Approximation` | `WGS84 Exact`

Method to calculate gravity, specified as:

Gravity Model Method	Output
`WGS84 Taylor Series` and `WGS84 Close Approximation`	Output only normal gravity (down in the NED coordinate system).
`WGS84 Exact`	Both normal and tangent gravity (down and north in the NED coordinate system).

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'WGS84 Taylor Series'` | `'WGS84 Close Approximation'` | `'WGS84 Exact'`
Default: `'WGS84 Taylor Series'`

Units — Units

`Metric (MKS)` (default) | `English`

Input and output units, specified as:

Units	Height	Gravity
`Metric (MKS)`	Meters	Meters per second squared
`English`	Feet	Feet per second squared

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English'`
Default: `'Metric (MKS)'`

Exclude Earth's atmosphere — Earth atmosphere

on (default) | off

- To exclude the mass of the atmosphere for the Earth gravitational field, select this check box.
- To include the mass of the atmosphere for the Earth gravitation field, clear this check box.

Dependencies

To enable this check box, set **Type of gravity model** to `Type of gravity model WGS84 Close Approximation` or `WGS84 Exact`.

Programmatic Use
Block Parameter: `no_atmos`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Precessing reference frame — Precessing reference frame

on (default) | off

- To calculate the velocity of the Earth using the International Astronomical Union (IAU) value of the Earth's angular velocity and the precession rate in right ascension, select this check box.
- To calculate the velocity of the Earth using the angular velocity of the standard Earth rotating at a constant angular velocity, clear this check box.

To obtain the precession rate in right ascension, the block calculates Julian centuries from Epoch J2000.0 using **Month**, **Day**, and **Year**.

Dependencies

- To enable this check box, set **Type of gravity model** to `Type of gravity model WGS84 Close Approximation` or `WGS84 Exact`.
- Clearing this check box disables the **Input Julian date** parameter and the **JD** input port.

Programmatic Use
Block Parameter: `precessing`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Input Julian date — Julian date

off (default) | on

- To specify the Julian date for the block with an input port, select this check box.
- To calculate the Julian date using the values of **Month**, **Day**, and **Year**, clear this check box. The year must be after January 1, 2000 (2451545).

Dependencies

- To enable the **JD** port, select this check box.

Programmatic Use
Block Parameter: `jd_loc`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Month — Month

`January` (default) | `February` | `March` | `April` | `May` | `June` | `July` | `August` | `September` | `October` | `November` | `December`

Month to calculate Julian centuries from Epoch J2000.0.

Dependencies

To enable this parameter:

- Set **Type of gravity model** to `WGS84 Close Approximation` or `WGS84 Exact`.
- Select **Precessing reference frame**.

To disable this parameter, select **Input Julian date**.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Day

10 (default) | 1 to 31

Day to calculate Julian centuries from Epoch J2000.0.

Dependencies

To enable this parameter:

- Set **Type of gravity model** to `WGS84 Close Approximation` or `WGS84 Exact`.
- Select **Precessing reference frame**.

To disable this parameter, select **Input Julian date**.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` to `'31'`
Default: `'10'`

Year — Year

2004 (default) | any year

Year to calculate Julian centuries from Epoch J2000.0. The year must be 2000 or greater.

Dependencies

To enable this parameter:

- Set **Type of gravity model** to `WGS84 Close Approximation` or `WGS84 Exact`.
- Select **Precessing reference frame**.
- To disable this parameter, select **Input Julian date**.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: any year
Default: `'2004'`

No centrifugal effects — Centrifugal effects

`on` (default) | `off`

- To base calculated gravity on pure attraction resulting from the normal gravitational potential, select this check box.
- To enable the calculated gravity to include the centrifugal force resulting from the Earth's angular velocity, clear this check box.

This option is available only with **Type of gravity model WGS84 Close Approximation** or **WGS84 Exact**.

Programmatic Use
Block Parameter: `no_centrifugal`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Action for out-of-range input — Out-of-range block behavior

`Warning` (default) | `Error` | `None`

Out-of-range block behavior, specified as follows.

Value	Description
`None`	No action. The block imposes upper and lower limits on an input signal.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues. For Accelerator and Rapid Accelerator modes, setting the action to `Warning` has no effect and the model behaves as though the action is set to `None`.
`Error`	MATLAB returns an exception, model simulation stops. For Accelerator and Rapid Accelerator modes, setting the action to `Error` has no effect and the model behaves as though the action is set to `None`.

Programmatic Use
Block Parameter: `action`

Type: character vector
Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Version History

Introduced before R2006a

References

[1] "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems." NIMA TR8350.2.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

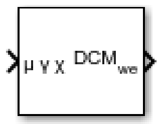
COESA Atmosphere Model

Topics

"NASA HL-20 Lifting Body Airframe" on page 3-14

Wind Angles to Direction Cosine Matrix

Convert wind angles to direction cosine matrix



Libraries:
Aerospace Blockset / Utilities / Axes Transformations

Description

The Wind Angles to Direction Cosine Matrix block converts three wind rotation angles into a 3-by-3 direction cosine matrix (DCM). The DCM matrix performs the coordinate transformation of a vector in earth axes (ox_0, oy_0, oz_0) into a vector in wind axes (ox_3, oy_3, oz_3). For more information on the direction cosine matrix, see “Algorithms” on page 5-1291.

This implementation generates a flight path angle that lies between ±90 degrees, and bank and heading angles that lie between ±180 degrees.

Ports

Input

μ γ χ — Wind rotation angles
3-by-1 vector

Wind rotation angles, specified as a 3-by-1 vector, in radians.

Data Types: `double`

Output

$\mathbf{DCM_{we}}$ — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, returned as a 3-by-3 matrix.

Data Types: `double`

Algorithms

The DCM matrix performs the coordinate transformation of a vector in earth axes (ox_0, oy_0, oz_0) into a vector in wind axes (ox_3, oy_3, oz_3). The order of the axis rotations required to bring this about is:

1. A rotation about oz_0 through the heading angle (χ) to axes (ox_1, oy_1, oz_1)
2. A rotation about oy_1 through the flight path angle (γ) to axes (ox_2, oy_2, oz_2)
3. A rotation about ox_2 through the bank angle (μ) to axes (ox_3, oy_3, oz_3)

$$\begin{bmatrix} ox_3 \\ oy_3 \\ oz_3 \end{bmatrix} = DCM_{we} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

$$\begin{bmatrix} ox_3 \\ oy_3 \\ oz_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\mu & \sin\mu \\ 0 & -\sin\mu & \cos\mu \end{bmatrix} \begin{bmatrix} \cos\gamma & 0 & -\sin\gamma \\ 0 & 1 & 0 \\ \sin\gamma & 0 & \cos\gamma \end{bmatrix} \begin{bmatrix} \cos\chi & \sin\chi & 0 \\ -\sin\chi & \cos\chi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ox_0 \\ oy_0 \\ oz_0 \end{bmatrix}$$

Combining the three axis transformation matrices defines the following DCM:

$$DCM_{we} = \begin{bmatrix} \cos\gamma\cos\chi & \cos\gamma\sin\chi & -\sin\gamma \\ (\sin\mu\sin\gamma\cos\chi - \cos\mu\sin\chi) & (\sin\mu\sin\gamma\sin\chi + \cos\mu\cos\chi) & \sin\mu\cos\gamma \\ (\cos\mu\sin\gamma\cos\chi + \sin\mu\sin\chi) & (\cos\mu\sin\gamma\sin\chi - \sin\mu\cos\chi & \cos\mu\cos\gamma \end{bmatrix}$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

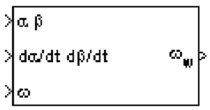
Generate C and C++ code using Simulink® Coder™.

See Also

Direction Cosine Matrix Body to Wind | Direction Cosine Matrix to Rotation Angles | Direction Cosine Matrix to Wind Angles | Rotation Angles to Direction Cosine Matrix

Wind Angular Rates

Calculate wind angular rates from body angular rates, angle of attack, sideslip angle, rate of change of angle of attack, and rate of change of sideslip



Libraries:
Aerospace Blockset / Flight Parameters

Description

The Wind Angular Rates block supports the equations of motion in wind-fixed frame models by calculating the wind-fixed angular rates (p_w, q_w, r_w). For more information on the equation used for the calculation, see “Algorithms” on page 5-1294.

Ports

Input

α β — Angles of attack and sideslip
2-by-1 vector

Angle of attack and sideslip, specified as a 2-by-1 vector, in radians.

Data Types: `double`

dα/dt dβ/dt — Rates of change
2-by-1 vector

Rate of change of the angle of attack and rate of change of the sideslip, specified as a 2-by-1 vector, in radians per second.

Data Types: `double`

ω — Body angular rates
three-element vector

Body angular rates, specified as a three-element vector, in radians per second.

Data Types: `double`

Output

ω_w — Wind angular rates
three-element vector

Wind angular rates, returned as a three-element vector, in radians per second.

Data Types: `double`

Algorithms

The body-fixed angular rates (p_b, q_b, r_b), angle of attack (α), sideslip angle (β), rate of change of angle of attack ($\dot{\alpha}$), and rate of change of sideslip ($\dot{\beta}$) are related to the wind-fixed angular rates as illustrated in the following equation:

$$\begin{bmatrix} p_w \\ q_w \\ r_w \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\beta & \sin\beta & \sin\alpha\cos\beta \\ -\cos\alpha\sin\beta & \cos\beta & -\sin\alpha\sin\beta \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \begin{bmatrix} p_b - \dot{\beta}\sin\alpha \\ q_b - \dot{\alpha} \\ r_b + \dot{\beta}\cos\alpha \end{bmatrix}$$

Version History

Introduced before R2006a

Extended Capabilities

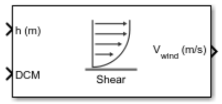
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

3DOF (Body Axes) | 6DOF Wind (Quaternion) | 6DOF Wind (Wind Angles) | Custom Variable Mass 3DOF (Body Axes) | Custom Variable Mass 6DOF Wind (Quaternion) | Custom Variable Mass 6DOF Wind (Wind Angles) | Simple Variable Mass 3DOF (Body Axes) | Simple Variable Mass 6DOF Wind (Quaternion) | Simple Variable Mass 6DOF Wind (Wind Angles)

Wind Shear Model

Calculate wind shear conditions



Libraries:
Aerospace Blockset / Environment / Wind

Description

The Wind Shear Model block adds wind shear to the aerospace model. This implementation is based on the mathematical representation in the Military Specification MIL-F-8785C [1].

Ports

Input

h — Altitude
scalar

Altitude, specified as a scalar in specified units.

Data Types: `double`

DCM — Direction cosine matrix
3-by-3 matrix

Direction cosine matrix, specified as a 3-by-3 matrix representing the flat Earth coordinates to body-fixed axis coordinates.

Data Types: `double`

Output

V_{wind} — Mean wind speed
three-element vector

Mean wind speed, returned as a three-element vector in the same body coordinate reference as the **DCM** input, in specified units.

Data Types: `double`

Parameters

Units — Wind shear units

`Metric (MKS)` (default) | `English (Velocity in ft/s)` | `English (Velocity in kts)`

Wind shear units, specified as:

Units	Wind	Altitude
`Metric (MKS)`	Meters/second	Meters
`English (Velocity in ft/s)`	Feet/second	Feet
`English (Velocity in kts)`	Knots	Feet

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (Velocity in ft/s)'` | `'English (Velocity in kts)'`
Default: `'Metric (MKS)'`

Flight phase — Flight phase

`Category C - Terminal Flight Phase` (default) | `Other`

Flight phase, specified as:

- `Category C - Terminal Flight Phase`, as specified by Military Specification MIL-F-8785C.
- `Other`

Programmatic Use
Block Parameter: `phase`
Type: character vector
Values: `'Category C - Terminal Flight Phase'` | `'Other'`
Default: `'Category C - Terminal Flight Phase'`

Wind speed at 20 ft altitude (kts) — Wind speed

`15` (default) | scalar

Measured wind speed at a height of 6 m (20 ft) above the ground, specified as a scalar.

Programmatic Use
Block Parameter: `W_20`
Type: character vector
Values: `scalar`
Default: `'15'`

Wind direction at 20 ft altitude (degrees clockwise from north) — Wind direction

`0` (default) | scalar

Wind direction at a height of 6 m (20 ft) above the ground, specified as a scalar, in degrees clockwise from the direction of the Earth *x*-axis (north). The wind direction is defined as the direction from which the wind is coming.

Programmatic Use
Block Parameter: `Wdeg`
Type: character vector
Values: `scalar`
Default: `'0'`

Algorithms

The magnitude of the wind shear is given by the following equation for the mean wind profile as a function of altitude and the measured wind speed at 20 feet (6 m) above the ground.

$$u_w = W_{20}\frac{\ln\left(\frac{h}{z_0}\right)}{\ln\left(\frac{20}{z_0}\right)},\ 3ft < h < 1000ft$$

where u_w is the mean wind speed, W_{20} is the measured wind speed at an altitude of 20 feet, h is the altitude, and z_0 is a constant equal to 0.15 feet for Category C flight phases and 2.0 feet for all other flight phases. Category C flight phases are defined in reference [1] to be terminal flight phases, which include takeoff, approach, and landing.

The resultant mean wind speed in the flat Earth axis frame is changed to body-fixed axis coordinates by multiplying by the direction cosine matrix (DCM) input to the block. The block output is the mean wind speed in the body-fixed axis.

Version History

Introduced before R2006a

References

[1] U.S. Military Specification MIL-F-8785C, November 5, 1980.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

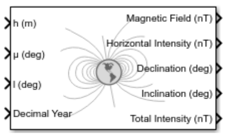
Discrete Wind Gust Model | Dryden Wind Turbulence Model (Continuous) | Dryden Wind Turbulence Model (Discrete) | Von Karman Wind Turbulence Model (Continuous)

Topics

“NASA HL-20 Lifting Body Airframe” on page 3-14

World Magnetic Model

Calculate Earth's magnetic field at specific location and time using World Magnetic Model



Libraries:
Aerospace Blockset / Environment / Gravity

Description

The World Magnetic Model block implements the mathematical representation of the National Geospatial Intelligence Agency (NGA) World Magnetic Model. The World Magnetic Model block calculates the Earth magnetic field vector, horizontal intensity, declination, inclination, and total intensity at a specified location and time. The reference frame is north-east-down (NED).

Note Use this block to model the Earth magnetic field between altitudes of -1,000 m to 850,000 m meters.

Limitations

All specifications have these limitations:

- The internal calculation of decimal year does not take into account local time or leap seconds.
- The specifications describe only the long-wavelength spatial magnetic fluctuations in the Earth's core. Intermediate and short-wavelength fluctuations, contributed from the crustal field (the mantle and crust), are not included. Also, the substantial fluctuations of the geomagnetic field, which occur constantly during magnetic storms and almost constantly in the disturbance field (auroral zones), are not included.
- This block has the limitations of the World Magnetic Model (WMM). WMM2020 is valid between -1km and 850km, as outlined in the World Magnetic Model 2020 Technical Report.

In addition, each specification has these limitations:

- WMM2015v2 supersedes WMM2015(v1). Consider replacing WMM2015(v1) with WMM2015v2 when used for navigation and other systems. WMM2015v2 was released by National Oceanic and Atmospheric Administration (NOAA) in February 2019 to correct performance degradation issues in the Arctic region for January 1, 2015, to December 31, 2019. Therefore, it is still acceptable to use WMM2015(v1) in systems below 55 degrees latitude in the Northern hemisphere.
- The WMM2020 specification produces data that is reliable five years after the epoch of the model, which is January 1, 2020.
- The WMM2015 specification produces data that is reliable five years after the epoch of the model, which is January 1, 2015.
- The WMM2010 specification produces data that is reliable five years after the epoch of the model, which is January 1, 2010.
- The WMM2005 specification produces data that is reliable five years after the epoch of the model, which is January 1, 2005.

- The WMM2000 specification produces data that is reliable five years after the epoch of the model, which is January 1, 2000.

Ports

Input

h — Height
scalar

Height, specified as a scalar, in selected units.

Data Types: `double`

μ (deg) — Latitude
scalar

Latitude, specified as a scalar, in degrees. If latitude is out of range, the block wraps it to be within the range when **Action for out-of-range** input is set to `None` or `Warning`. It does not wrap when **Action for out-of-range** is set to `Error`.

Data Types: `double`

l (deg) — Longitude
scalar

Longitude, specified as a scalar, in degrees. If longitude is out of range, the block wraps it to be within the range when **Action for out-of-range** input is set to `None` or `Warning`. It does not wrap when **Action for out-of-range** is set to `Error`.

Data Types: `double`

Decimal Year — Desired year
scalar

Desired year in a decimal format to include any fraction of the year that has already passed. The value is the current year plus the number of days that have passed in this year divided by 365.

For example, to calculate the decimal year, `dyear`, for March 21, 2015:

```
dyear=decyear('21-March-2015','dd-mmm-yyyy')
```

```
dyear =
   2.0152e+03
```

Data Types: `double`

Output

Magnetic Field (nT) — Magnetic field
vector

Magnetic field, returned as a vector, in selected units.

Data Types: `double`

Horizontal Intensity (nT) — Horizontal intensity
scalar

Horizontal intensity, returned as a scalar, in specified units.

Data Types: `double`

Declination (deg) — Declination
scalar

Declination, returned as a scalar, in degrees.

Data Types: `double`

Inclination (deg) — Inclination
scalar

Inclination, returned as a scalar, in degrees.

Data Types: `double`

Total Intensity (nT) — Total intensity
scalar

Total intensity, returned as a scalar, in selected units.

Data Types: `double`

Parameters

WMM coefficients — World Magnetic Model coefficient file

`WMM2020 (2020-2025)` (default) | `WMM2015 V2 (2015-2020)` | `WMM2000 (2000-2005)` | `WMM2005 (2005-2010)` | `WMM2010 (2010-2015)` | `WMM2015 V1 (2015-2020)` | `Custom`

World Magnetic Model coefficient file, selected from the list.

- `WMM2000 (2000-2005)` — World Magnetic Model 2000 coefficient file
- `WMM2005 (2005-2010)` — World Magnetic Model 2005 coefficient file
- `WMM2010 (2010-2015)` — World Magnetic Model 2010 coefficient file
- `WMM2015 V1 (2015-2020)` — World Magnetic Model 2015 (v1) coefficient file
- `WMM2015 V2 (2015-2020)` — World Magnetic Model 2015 (v2) coefficient file
- `WMM2020 (2020-2025)` — World Magnetic Model 2020 coefficient file
- `Custom` — Specify your own World Magnetic Model coefficient file. You can download a World Magnetic Model coefficient file from The World Magnetic Model.

Dependencies

Selecting `Custom` enables the **Custom .COF file** parameter.

Programmatic Use
Block Parameter: `model`
Type: character vector
Values: `'WMM2020 (2020-2025)'` | `'WMM2015 V2 (2015-2020)'` | `'WMM2000 (2000-2005)'` | `'WMM2005 (2005-2010)'` | `'WMM2010 (2010-2015)'` | `'WMM2015 V1 (2015-2020)'` | `'Custom'`
Default: `'WMM2020 (2020-2025)'`

Custom .COF file — Custom World Magnetic Model coefficient file

'WMM2020COF' (default) | any coefficient file name

World Magnetic Model coefficient file, downloaded from The World Magnetic Model. For example, if you want to download a coefficient file not yet listed in the **WMM coefficients** list.

Dependencies

To enable this parameter, select Custom for the **WMM coefficients** parameter.

Programmatic Use
Block Parameter: customFile
Type: character vector
Values: 'WMM2020.COF' | any coefficient file name
Default: 'WMM2020.COF'

Units — Input and output units

Metric (MKS) (default) | English

Input and output units:

Units	Height	Magnetic Field	Horizontal Intensity	Total Intensity
Metric (MKS)	Meters	Nanotesla	Nanotesla	Nanotesla
English	Feet	Nanogauss	Nanogauss	Nanogauss

Programmatic Use
Block Parameter: units
Type: character vector
Values: 'Metric (MKS)' | 'English'
Default: 'Metric (MKS)'

Input decimal year — Input decimal year

on (default) | off

- To specify the decimal year with an input port for the World Magnetic Model 2015 block, select this check box
- To specify the decimal year using the values of **Month**, **Day**, and **Year**, clear this check box.

Programmatic Use
Block Parameter: time_in
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Month — Input month

January (default) | February | March | April | May | June | July | August | September | October | November | December

Month to calculate decimal year.

Dependencies

To enable this parameter, select **Input decimal year**.

Programmatic Use
Block Parameter: `month`
Type: character vector
Values: `'January'` | `'February'` | `'March'` | `'April'` | `'May'` | `'June'` | `'July'` | `'August'` | `'September'` | `'October'` | `'November'` | `'December'`
Default: `'January'`

Day — Input day

1 (default) | 1 to 31

Day to calculate decimal year.

Dependencies

To enable this parameter, select **Input decimal year**.

Programmatic Use
Block Parameter: `day`
Type: character vector
Values: `'1'` to `'31'`
Default: `'1'`

Year — Input year

2020 (default) | any year

Year to calculate decimal year.

Dependencies

To enable this parameter, select **Input decimal year**.

Programmatic Use
Block Parameter: `year`
Type: character vector
Values: any year
Default: `'2020'`

Action for out-of-range input — Out-of-range action

`Error` (default) | `Warning` | `None`

Out-of-range block behavior, specified as follows.

Action	Description
`None`	No action.
`Warning`	Warning in the Diagnostic Viewer, model simulation continues.
`Error` (default)	Error in the Diagnostic Viewer, model simulation stops.

If longitude or latitude is out of range, the block wraps it to be within the range when **Action for out-of-range** input is set to `None` or `Warning`. It does not wrap when **Action for out-of-range** is set to `Error`.

The World Magnetic Model block accepts out-of-range height inputs (less than -1000 m or greater than 850000 m) when **Action for out-of-range** is set to `None` or `Warning`. However, the block output might not be accurate or reliable for these values. The World Magnetic Model is valid only between -1000 m and 850000 m.

Programmatic Use
Block Parameter: `action`
Type: character vector
Values: `'Error'` | `'Warning'` | `'None'`
Default: `'Error'`

Output horizontal intensity — Output horizontal intensity

on (default) | `off`

To output the horizontal intensity value, select this check box. Otherwise, clear this check box.

Dependencies

To enable the **Horizontal Intensity** output port, select this check box.

Programmatic Use
Block Parameter: `h_out`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Output declination — Output declination

on (default) | `off`

To output the declination, the angle between true north and the magnetic field vector (positive eastwards), select this check box. Otherwise, clear this check box.

Dependencies

To enable the **Declination** output port, select this check box.

Programmatic Use
Block Parameter: `dec_out`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Output inclination — Output inclination

on (default) | `off`

To output the inclination, the angle between the horizontal plane and the magnetic field vector (positive downwards), select this check box. Otherwise, clear this check box.

Dependencies

To enable the **Inclination** output port, select this check box.

Programmatic Use

Block Parameter: `inc_out`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Output total intensity — Output total intensity

on (default) | `off`

To output the total intensity, select this check box. Otherwise, clear this check box.

Dependencies

To enable the **Total Intensity** output port, select this check box.

Programmatic Use

Block Parameter: `ti_out`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

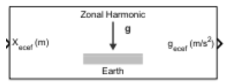
International Geomagnetic Reference Field | `decyear`

External Websites

The World Magnetic Model

Zonal Harmonic Gravity Model

Calculate zonal harmonic representation of planetary gravity



Libraries:
Aerospace Blockset / Environment / Gravity

Description

The Zonal Harmonic Gravity Model block calculates the zonal harmonic representation of planetary gravity at a specific location based on planetary gravitational potential. This block provides a convenient way to describe the gravitational field of a planet outside its surface.

By default, the block uses the fourth order zonal coefficient for Earth to calculate the zonal harmonic gravity. It also allows you to specify the second or third zonal coefficient.

For information on the planetary parameter values for each planet in the block implementation, see “Algorithms” on page 5-1308.

Limitations

The block excludes the centrifugal effects of planetary rotation and the effects of a precessing reference frame.

Ports

Input

X_{ecef} — Planet-centered planet-fixed coordinates
m-by-3 matrix

Planet-centered planet-fixed coordinates, specified as an *m*-by-3 matrix, from the center of the planet in the selected length units. If **Planet model** has a value of `Earth`, this matrix contains Earth-centered Earth-fixed (ECEF) coordinates.

Data Types: `double`

Output

g_{ecef} — Gravity values
m-by-3 array

Gravity values, returned as an *m*-by-3 array, in the *x*-axis, *y*-axis and *z*-axis of the planet-centered planet-fixed coordinates, in the selected length units per second squared.

Data Types: `double`

Parameters

Units — Input units

Metric (MKS) (default) | English

Input units, specified as:

Units	Position	Equatorial Radius	Gravitational Parameter
Metric (MKS)	Meters	Meters	Meters cubed per second squared
English	Feet	Feet	Feet cubed per second squared

Programmatic Use
Block Parameter: units
Type: character vector
Values: 'Metric (MKS)' | 'English'
Default: 'Metric (MKS)'

Degree — Degree of harmonic model

4 (default) | 2 | 3

Degree of harmonic model, specified as.

- 2 — Second degree, J2. Most significant or largest spherical harmonic term, which accounts for the oblateness of a planet.
- 3 — Third degree, J3.
- 4 — Fourth degree, J4 (default).

Programmatic Use
Block Parameter: degree
Type: character vector
Values: '2' | '3' | '4'
Default: '4'

Action for out-of-range input — Out-of-range input behavior

Warning (default) | ErrorNone

Out-of-range input behavior, specified as:

Value	Description
None	No action.
Warning	Warning in the Diagnostic Viewer, model simulation continues.
Error	MATLAB returns an exception, model simulation stops.

Programmatic Use
Block Parameter: action
Type: character vector

Values: `'None'` | `'Warning'` | `'Error'`
Default: `'Warning'`

Planet model — Planetary model

`Mercury` (default) | `Venus` | `Earth` | `Moon` | `Mars` | `Jupiter` | `Saturn` | `Uranus` | `Neptune` | `Custom`

Planetary model, specified as `Mercury`, `Venus`, `Earth`, `Moon`, `Mars`, `Jupiter`, `Saturn`, `Uranus`, `Neptune`, or `Custom`.

Selecting `Custom` enables you to specify your own planetary model.

- Selecting `Mercury`, `Venus`, `Moon`, `Uranus`, or `Neptune` limits the degree to `2`.
- Selecting `Mars` limits the degree to `3`.

Dependencies

Selecting `Custom` enables the **Equatorial radius**, **Gravitational parameter** and **J values rate** parameters.

Programmatic Use
Block Parameter: `ptype`
Type: character vector
Values: `'Mercury'` | `'Venus'` | `'Earth'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Custom'`
Default: `'Earth'`

Equatorial radius — Planetary equatorial radius

`6378136.3` (default) | scalar

Planetary equatorial radius, specified as a scalar, in the length units that the **Units** parameter defines.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `R`
Type: character vector
Values: scalar
Default: `'6378136.3'`

Gravitational parameter — Planetary gravitational parameter

`398600441500000` (default) | scalar

Planetary gravitational parameter, specified as a scalar, in the length units cubed per second squared that the **Units** parameter defines.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `GM`
Type: character vector
Values: scalar
Default: `'398600441500000'`

J values — Zonal harmonic coefficients

`[1.0826269e-03 -2.5323000e-06 -1.6204000e-06]` (default) | 3-element array

Zonal harmonic coefficient, specified as a 3-element array.

Dependencies

To enable this parameter, set **Planet model** to `Custom`.

Programmatic Use
Block Parameter: `jvalue`
Type: character vector
Values: scalar
Default: `'[1.0826269e-03 -2.5323000e-06 -1.6204000e-06]'`

Algorithms

This block is implemented using the following planetary parameter values for each planet:

Planet	Equatorial Radius (Re) in Meters	Gravitational Parameter (GM) in m^3/s^2	Zonal Harmonic Coefficients (J Values)
Earth	6378.1363e3	3.986004415e14	[0.0010826269 -0.0000025323 -0.0000016204]
Jupiter	71492e3	1.268e17	[0.01475 0 -0.00058]
Mars	3397.2e3	4.305e13	[0.001964 0.000036]
Mercury	2439.0e3	2.2032e13	0.00006
Moon	1738.0e3	4902.799e9	0.0002027
Neptune	24764e3	6.809e15	0.004
Saturn	60268e3	3.794e16	[0.01645 0 -0.001]
Uranus	25559e3	5.794e15	0.012
Venus	6052.0e3	3.257e14	0.000027

Version History

Introduced in R2009b

References

[1] Vallado, David, *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill, 1997.

[2] Fortescue, P., J. Stark, G. Swinerd, eds.. *Spacecraft Systems Engineering*, 3d ed. West Sussex: Wiley & Sons, 2003.

[3] Tewari, A. Boston: *Atmospheric and Space Flight Dynamics Modeling and Simulation with MATLAB and Simulink*. Boston: Birkhäuser, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Centrifugal Effect Model | Spherical Harmonic Gravity Model

6

Functions

asbFlightControlAnalysis

Start flight control analysis template

Syntax

```
asbFlightControlAnalysis()
asbFlightControlAnalysis(configuration)
asbFlightControlAnalysis(configuration,modelToAnalyze)
asbFlightControlAnalysis(configuration,modelToAnalyze,airframe)
```

Description

`asbFlightControlAnalysis()` creates a flight control analysis template for a 3DOF configuration.

`asbFlightControlAnalysis(configuration)` creates a flight control analysis template for a specified configuration.

`asbFlightControlAnalysis(configuration,modelToAnalyze)` creates a flight control analysis model with the specified model name.

`asbFlightControlAnalysis(configuration,modelToAnalyze,airframe)` creates a flight control analysis template for a specified airframe model.

Examples

Start Flight Control Analysis Template for 3DOF Configuration

Start default flight control analysis template for 3DOF configuration.

```
asbFlightControlAnalysis
```

Start Flight Control Analysis Template for 6DOF Configuration

Start default flight control analysis template for 6DOF configuration.

```
asbFlightControlAnalysis('6DOF')
```

Start Flight Control Analysis Template Using a Different Airframe Model

Start the 3DOF flight control analysis template `SkyHoggAnalysisModel` and trim the model around the `opSpecDefault` operating point specification object. The example then linearizes the airframe model around the `opTrim` operating point and calculates the short- and long-period (phugoid) mode characteristics of `linSys`.

```
asbFlightControlAnalysis('3DOF', 'SkyHoggAnalysisModel');
opSpecDefault = SkyHogg3DOFOpSpec('SkyHoggAnalysisModel');
opTrim = trimAirframe('SkyHoggAnalysisModel', opSpecDefault);
```

```
linSys = linearizeAirframe('SkyHoggAnalysisModel', opTrim)
flyingQual = computeLongitudinalFlyingQualities('SkyHoggAnalysisModel', linSys)
```

Input Arguments

`configuration` — Configuration for flight control analysis
`'3DOF'` (default) | `'6DOF'`

Configuration for flight control analysis.

Data Types: `char` | `string`

`modelToAnalyze` — Name for flight control analysis model being created
model name

Name for flight control analysis model being created.

Data Types: `char` | `string`

`airframe` — Airframe to analyze
airframe subsystem specified as a block path (default) | airframe model specified as a model name

Airframe to analyze, specified as an airframe model name (inserted as a referenced model). Otherwise, the subsystem must be loaded.

Data Types: `char` | `string`

Version History

Introduced in R2018b

See Also

`computeLateralDirectionalFlyingQualities` | `computeLongitudinalFlyingQualities` | `linearizeAirframe` | `trimAirframe`

Topics

"Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles" on page 2-65

ASim3dActor

Abstract class to use as a base class for user-defined Unreal Engine C++ or blueprint actors

Description

`ASim3dActor` is an abstract class that you can use as a base class for user-defined Unreal Engine C++ or blueprint actors.

The base classes are inherently synchronized during co-simulation with a Simulink model. Additionally, the Simulation 3D Actor Transform Set block can control the base class. To extend behavior of `ASim3dActor`, you can use the message interface functions to override the class methods so they send and receive messages to and from a model.

`ASim3dActor` is included in the Aerospace Blockset Interface for Unreal Engine Projects. For information about the support package, see "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2.

Properties

`Translation` — Actor translation
1-by-3 (default) | number of parts per actor-by-3

This property is protected. It is used in the derived C++ class. Value is set by the Simulation 3D Actor Transform Set block.

Actor translation along world *X*-, *Y*, and *Z*- axes, respectively, in m. Array dimensions are number of parts per actor-by-3.

Data Types: `float`

`Rotation` — Actor rotation
1-by-3 (default) | number of parts per actor-by-3

This property is protected. It is used in the derived C++ class. Value is set by the Simulation 3D Actor Transform Set block.

Actor rotation across a [-pi/2, pi/2] range about world *X*-, *Y*, and *Z*- axes, respectively, in rad. Array dimensions are number of parts per actor-by-3.

Data Types: `float`

`Scale` — Actor scale
1-by-3 (default) | number of parts per actor-by-3

This property is protected. It is used in the derived C++ class. Value is set by the Simulation 3D Actor Transform Set block.

Actor scale. Array dimensions are number of parts per actor-by-3.

Data Types: `float`

Object Functions

Sim3dSetup	C++ method that sets up actor in Unreal Engine 3D simulation
Sim3dStep	C++ method that steps actor in Unreal Engine 3D simulation
Sim3dRelease	C++ method that releases actor in Unreal Engine 3D simulation

Version History

Introduced in R2021b

See Also

`StartSimulation3DMessageReader` | `ReadSimulation3DMessage` | `StopSimulation3DMessageReader` | `StartSimulation3DMessageWriter` | `WriteSimulation3DMessage` | `StopSimulation3DMessageWriter`

External Websites

Unreal Engine 4 Documentation

computeLateralDirectionalFlyingQualities

Calculate dutch roll mode, roll mode, and spiral mode characteristics of state-space model

Syntax

computeLateralDirectionalFlyingQualities(modelToAnalyze)
lonFQOut = computeLateralDirectionalFlyingQualities(modelToAnalyze,linSys)
lonFQOut = computeLateralDirectionalFlyingQualities(modelToAnalyze,linSys,generatePlots)
[lonFQOut,varNameOut] = computeLateralDirectionalFlyingQualities(___ ,Name,Value)

Description

computeLateralDirectionalFlyingQualities(modelToAnalyze) calculates the lateral-directional flying qualities (dutch roll mode, roll mode, and spiral mode) characteristics using the linear system state-space model selected in the input dialog window and compares the results against the specified source document requirements.

lonFQOut = computeLateralDirectionalFlyingQualities(modelToAnalyze,linSys) calculates lateral-directional flying quality characteristics (dutch roll mode, roll mode, and spiral mode) using the linear system state-space model provided as an input to the function.

lonFQOut = computeLateralDirectionalFlyingQualities(modelToAnalyze,linSys,generatePlots) displays the pole-zero map for the linear system state-space model.

[lonFQOut,varNameOut] = computeLateralDirectionalFlyingQualities(___ ,Name,Value) returns the output results structure variable name, varNameOut, for the input argument combination in the previous syntax, according to the Name,Value arguments.

Examples

Calculate Lateral-Directional Flying Qualities of Simulink Aircraft Model

Calculate the lateral-directional flying qualities of a Simulink aircraft model.

```
asbFlightControlAnalysis('6DOF', 'DehavillandBeaverAnalysisModel');
opSpecDefault = DehavillandBeaver6DOFOpSpec('DehavillandBeaverAnalysisModel');
opTrim = trimAirframe('DehavillandBeaverAnalysisModel', opSpecDefault);
linSys = linearizeAirframe('DehavillandBeaverAnalysisModel', opTrim);
latFlyingQual = computeLateralDirectionalFlyingQualities('DehavillandBeaverAnalysisModel', linSys)
```

```
Operating point search report:
---------------------------------

 Operating point search report for the Model DehavillandBeaverAnalysisModel.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) phi
      x:         0.021      dx:     -1.12e-20 (0)
(2.) theta
      x:        0.0653      dx:      3.91e-22 (0)
```

```
(3.) psi
      x:             0      dx:      -1.7e-20 (0)
(4.) p
      x:         -1e-20     dx:     -7.37e-12 (0)
(5.) q
      x:      3.52e-23      dx:      3.42e-10 (0)
(6.) r
      x:     -1.69e-20      dx:      -1.2e-11 (0)
(7.) U
      x:          67.3      dx:      1.79e-13 (0)
(8.) v
      x:        0.0927      dx:     -4.63e-11 (0)
(9.) w
      x:           4.4      dx:      2.02e-11 (0)
(10.) Xe
      x:     -3.86e-13      dx:          67.5
(11.) Ye
      x:     -1.18e-12      dx:      4.21e-12 (0)
(12.) Ze
      x:      -2.2e+03      dx:      5.97e-11 (0)
(13.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on angular rates/Hpgw/
      x:             0      dx:             0
      x:             0      dx:             0
(14.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on angular rates/Hqgw/
      x:             0      dx:             0
      x:             0      dx:             0
(15.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on angular rates/Hrgw/
      x:             0      dx:             0
      x:             0      dx:             0
(16.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on velocities/Hugw(s)/
      x:             0      dx:             0
      x:             0      dx:             0
(17.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on velocities/Hvgw(s)/
      x:             0      dx:             0
      x:             0      dx:             0
(18.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on velocities/Hvgw(s)/
      x:             0      dx:             0
      x:             0      dx:             0
(19.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on velocities/Hwgw(s)/
      x:     -8.13e-14      dx:             0
      x:      5.37e-15      dx:             0
(20.) DehavillandBeaverAnalysisModel/Environment Model/Dryden Wind Turbulence Model  (Continuous (+q +r))/Filters on velocities/Hwgw(s)/
      x:             0      dx:             0
      x:             0      dx:             0

Inputs:
----------
(1.) DehavillandBeaverAnalysisModel/AileronCmd
      u:       0.00234    [-0.524 0.524]
(2.) DehavillandBeaverAnalysisModel/ElevatorCmd
      u:        0.0239    [-0.524 0.524]
(3.) DehavillandBeaverAnalysisModel/RudderCmd
      u:       -0.0377    [-1.05 1.05]
(4.) DehavillandBeaverAnalysisModel/ThrottleCmd
      u:         0.493    [0 1]

Outputs:
----------
(1.) DehavillandBeaverAnalysisModel/StatesOut
      y:     -3.86e-13    [-Inf Inf]
      y:     -1.18e-12    [-Inf Inf]
      y:      -2.2e+03    [-Inf Inf]
      y:         0.021    [-Inf Inf]
      y:        0.0653    [-Inf Inf]
      y:             0    [-Inf Inf]
      y:          67.3    [-Inf Inf]
      y:        0.0927    [-Inf Inf]
      y:           4.4    [-Inf Inf]
      y:        -1e-20    [-Inf Inf]
      y:      3.52e-23    [-Inf Inf]
      y:     -1.69e-20    [-Inf Inf]


latFlyingQual =

  struct with fields:

    DutchRollMode: [1×1 struct]
```

```
      RollMode: [1×1 struct]
    SpiralMode: [1×1 struct]
```

Calculate Lateral-Directional Flying Qualities of Aero.FixedWing Object

Calculate the lateral-directional flying qualities of an `Aero.FixedWing` object.

```
[aircraft, state] = astDehavillandBeaver();
linSys = linearize(aircraft, state)
latFlyingQual = computeLateralDirectionalFlyingQualities('', linSys)
```

```
linSys =

  A =
                 XN          XE          XD           U           V
   XN             0           0           0      0.9896           0
   XE             0           0           0           0           1
   XD             0           0           0     -0.1439           0
   U              0           0           0    -0.01339  -0.0004123
   V              0           0           0   -0.004288    -0.02862
   W              0           0           0     -0.1996    0.001044
   P              0           0           0  -0.0006608    -0.08777
   Q              0           0           0     0.03146   -0.002583
   R              0           0           0   0.0008302    0.003697
   Phi            0           0           0           0           0
   Theta          0           0           0           0           0
   Psi            0           0           0           0           0

                  W           P           Q           R         Phi
   XN        0.1439           0           0           0           0
   XE             0           0           0           0       6.475
   XD        0.9896           0           0           0   3.238e-05
   U          0.287           0     -0.2437           0      0.1845
   V      -0.006164     -0.2064           0      -44.39       9.621
   W         -1.262           0       43.92           0     -0.7921
   P      -0.001175      -5.218   -0.003787       1.771      -0.569
   Q        -0.1426  -1.697e-07      -2.947     -0.2721     -0.1121
   R      0.0001093     -0.8464      0.1728     -0.5366     0.02393
   Phi            0           1           0      0.1454   4.142e-22
   Theta          0           0           1           0   -2.99e-19
   Psi            0           0           0       1.011   2.878e-21

              Theta         Psi
   XN        -6.476  -0.0002227
   XE             0          45
   XD        -44.53   3.238e-05
   U          -9.89    0.008391
   V        0.03322       1.388
   W          1.043      0.1316
   P        0.00533    -0.08135
   Q        -0.0687      -0.023
   R      -0.005422    0.002902
   Phi    3.053e-19           0
   Theta          0           0
   Psi    4.394e-20           0

  B =
            Aileron       Flap   Elevator     Rudder  Propeller
```

```
  XN            0          0          0          0          0
  XE            0          0          0          0          0
  XD            0          0          0          0          0
  U             0     0.6608          0     0.3456      5.018
  V          -0.3          0          0       1.94          0
  W             0      -15.8     -4.068          0          0
  P        -7.019          0          0      0.491          0
  Q             0      2.163     -10.21          0          0
  R       -0.1925          0          0     -2.509          0
  Phi           0          0          0          0          0
  Theta         0          0          0          0          0
  Psi           0          0          0          0          0

 C =
           XN     XE     XD      U      V      W      P      Q      R
  XN        1      0      0      0      0      0      0      0      0
  XE        0      1      0      0      0      0      0      0      0
  XD        0      0      1      0      0      0      0      0      0
  U         0      0      0      1      0      0      0      0      0
  V         0      0      0      0      1      0      0      0      0
  W         0      0      0      0      0      1      0      0      0
  P         0      0      0      0      0      0      1      0      0
  Q         0      0      0      0      0      0      0      1      0
  R         0      0      0      0      0      0      0      0      1
  Phi       0      0      0      0      0      0      0      0      0
  Theta     0      0      0      0      0      0      0      0      0
  Psi       0      0      0      0      0      0      0      0      0

          Phi  Theta    Psi
  XN        0      0      0
  XE        0      0      0
  XD        0      0      0
  U         0      0      0
  V         0      0      0
  W         0      0      0
  P         0      0      0
  Q         0      0      0
  R         0      0      0
  Phi       1      0      0
  Theta     0      1      0
  Psi       0      0      1

 D =
          Aileron       Flap   Elevator     Rudder  Propeller
  XN            0          0          0          0          0
  XE            0          0          0          0          0
  XD            0          0          0          0          0
  U             0          0          0          0          0
  V             0          0          0          0          0
  W             0          0          0          0          0
  P             0          0          0          0          0
  Q             0          0          0          0          0
  R             0          0          0          0          0
  Phi           0          0          0          0          0
  Theta         0          0          0          0          0
  Psi           0          0          0          0          0

Continuous-time state-space model.
```

```
latFlyingQual = 

  struct with fields:

    DutchRollMode: [1×1 struct]
         RollMode: [1×1 struct]
       SpiralMode: [1×1 struct]
```

Input Arguments

`modelToAnalyze` — Model on which to perform flight control analysis
`''` (default) | model name

Model on which to perform flight control analysis using the linear state-space model `linSys`. To use a state-space model directly, set the model name to an empty string, `''`.

Data Types: `char` | `string`

`linSys` — State-space model object
`''` (default) | linear state-space model object name

State-space model object used to perform flight control analysis on `modelToAnalyze`. To create the state-space model from the input dialog menu, set `linSys` to an empty string, `''`. To create a valid state-space model, see `linearizeAirframe`.

The state-space model must have these state names:

- `U`
- `W`
- `Q`
- `theta`

Data Types: `char` | `string`

`generatePlots` — Display pole-zero map
`off` | `on`

Set to `on` to display pole-zero map for the linear system state-space model. Otherwise, set to `off`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'SourceDocument','MIL1797A'`

`SourceDocument` — Document for flying qualities requirements verification
`MIL8785C` (default) | `MIL1797A`

Document for flying qualities requirements verification, specified as:

- `MIL8785C` — Flying qualities of piloted airlines
- `MIL1797A` — Flying qualities of piloted aircraft

Data Types: `char` | `string`

`Level` — Flying qualities level
`Lowest` (default) | `All` | `1` | `2` | `3`

Flying qualities level, specified as:

- `Lowest` — Returns the verified requirements closest to level 1 for each requirement in the selected source document.
- `All` — Returns a `struct vector` with all requirement levels and their verification status.
- `1`, `2`, or `3` — Returns the desired requirement level, regardless of the verification status.

Data Types: `char` | `string`

Output Arguments

`lonFQOut` — Dutch roll, roll, and spiral lateral-directional flying qualities
structure vector

Dutch roll, roll, and spiral lateral-directional flying qualities, returned as a structure vector.

`varNameOut` — Output results structure
scalar string | `''`

If a linear system is selected through the input dialog, `varNameOut` returns the results structure variable name. Otherwise, `varNameOut` returns an empty string.

Limitations

This function requires the Simulink Control Design license.

Version History

Introduced in R2019a

See Also

`asbFlightControlAnalysis` | `computeLongitudinalFlyingQualities` | `linearizeAirframe` | `trimAirframe`

Topics

"Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles" on page 2-65

computeLongitudinalFlyingQualities

Calculate short-period and long-period (phugoid) mode characteristics of specified state-space model

Syntax

```
lonFQOut = computeLongitudinalFlyingQualities(modelToAnalyze)
lonFQOut = computeLongitudinalFlyingQualities(modelToAnalyze,linSys)
lonFQOut = computeLongitudinalFlyingQualities(modelToAnalyze,linSys,
generatePlots)
[lonFQOut,varNameOut] = computeLongitudinalFlyingQualities( ___ ,Name,Value)
```

Description

`lonFQOut = computeLongitudinalFlyingQualities(modelToAnalyze)` calculates longitudinal flying qualities (short-period and phugoid mode) using the linear system state-space model selected in the input dialog window and compares the results against the specified source document requirements.

`lonFQOut = computeLongitudinalFlyingQualities(modelToAnalyze,linSys)` calculates longitudinal flying qualities (short-period and phugoid mode) using the linear system state-space model selected in the input dialog window.

To create a usable state-space model, use the `linearizeAirframe` function.

`lonFQOut = computeLongitudinalFlyingQualities(modelToAnalyze,linSys, generatePlots)` calculates longitudinal flying qualities (short-period and phugoid mode) using linear system state-space model `linSys`.

`[lonFQOut,varNameOut] = computeLongitudinalFlyingQualities( ___ ,Name,Value)` returns the output results structure variable name, `varNameOut`, for the input argument combination in the previous syntax, according to the `Name,Value` arguments.

Examples

Calculate Longitudinal Flying Qualities of Simulink Aircraft Model

Calculate the longitudinal flying qualities of a Simulink aircraft model.

```
asbFlightControlAnalysis('3DOF', 'SkyHoggAnalysisModel');
opSpecDefault = SkyHogg3DOFOpSpec('SkyHoggAnalysisModel');
opTrim = trimAirframe('SkyHoggAnalysisModel', opSpecDefault);
linSys = linearizeAirframe('SkyHoggAnalysisModel', opTrim)
flyingQual = computeLongitudinalFlyingQualities('SkyHoggAnalysisModel', linSys)
```

```
 Operating point search report:
---------------------------------

 Operating point search report for the Model SkyHoggAnalysisModel.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
```

```
States:
----------
(1.) Xe
      x:     -4.45e-14      dx:           129
(2.) Ze
      x:        -2e+03      dx:     -1.69e-07 (0)
(3.) theta
      x:       0.00619      dx:             0 (0)
(4.) u
      x:           129      dx:      -7.9e-08 (0)
(5.) w
      x:         0.802      dx:     -5.24e-07 (0)
(6.) q
      x:             0      dx:     -8.41e-08 (0)

Inputs:
----------
(1.) SkyHoggAnalysisModel/ElevatorCmd
      u:        0.0125    [-0.349 0.349]
(2.) SkyHoggAnalysisModel/ThrottleCmd
      u:         0.929    [-Inf Inf]

Outputs:
----------
(1.) SkyHoggAnalysisModel/LonStatesBus
      y:     -4.45e-14    [-Inf Inf]
      y:        -2e+03    [-Inf Inf]
      y:       0.00619    [-Inf Inf]
      y:           129    [-Inf Inf]
      y:         0.802    [-Inf Inf]
      y:             0    [-Inf Inf]


linSys =

  A =
                 u          w          q      theta
   u      -0.05768    0.04733    -0.8016     -9.806
   w       -0.1149     -5.532      129.4   -0.06073
   q      0.001031    -0.1665          0          0
   theta         0          0          1          0

  B =
        ElevatorCmd  ThrottleCmd
   u         0.4828         0.36
   w          30.57            0
   q         -15.86            0
   theta          0            0

  C =
            u      w      q  theta
   q        0      0      1      0
   theta    0      0      0      1

  D =
        ElevatorCmd  ThrottleCmd
   q              0            0
   theta          0            0
```

```
Continuous-time state-space model.

flyingQual = 

  struct with fields:

        PhugoidMode: [1×1 struct]
    ShortPeriodMode: [1×1 struct]
```

Calculate Longitudinal Flying Qualities of Aero.FixedWing Object

Calculate the longitudinal flying qualities of an Aero.FixedWing object.

```
[aircraft, state] = astSkyHogg();
linSys = linearize(aircraft, state)
flyingQual = computeLongitudinalFlyingQualities('', linSys)
```

```
linSys =
 
  A = 
                XN         XD          U          W          Q      Theta
   XN            0          0     0.9999     0.0154          0  5.186e-05
   XD            0          0    -0.0154     0.9999          0     -1.719
   U             0          0   -0.04342     0.1119   -0.02653    -0.1712
   W             0          0    -0.1286     -4.082      1.719  -0.002637
   Q             0          0     0.1083     -7.037          0          0
   Theta         0          0          0          0          1          0
 
  B = 
         Elevator  Propeller
   XN           0          0
   XD           0          0
   U    -0.002381      8.837
   W      -0.2997          0
   Q       -8.908          0
   Theta        0          0
 
  C = 
           XN     XD      U      W      Q  Theta
   XN       1      0      0      0      0      0
   XD       0      1      0      0      0      0
   U        0      0      1      0      0      0
   W        0      0      0      1      0      0
   Q        0      0      0      0      1      0
   Theta    0      0      0      0      0      1
 
  D = 
         Elevator  Propeller
   XN           0          0
   XD           0          0
   U            0          0
   W            0          0
   Q            0          0
   Theta        0          0
 
Continuous-time state-space model.

flyingQual = 

  struct with fields:

        PhugoidMode: [1×1 struct]
    ShortPeriodMode: [1×1 struct]
```

Input Arguments

`modelToAnalyze` — Model on which to perform flight control analysis

'' (default) | model name

Model on which to perform flight control analysis using the linear state-space model `linSys`. To use a state-space model directly, set the model name to an empty string, `''`.

Data Types: `char` | `string`

`linSys` — Linear state-space model object
`''` (default) | linear state-space model object name

Linear state-space model object used to perform flight control analysis on `modelToAnalyze`. To create the state-space model from the input dialog menu, set `linSys` to an empty string, `''`. To create a valid state-space model, see `linearizeAirframe`.

The state-space model must have these state names:

- `U`
- `W`
- `Q`
- `theta`

Data Types: `char` | `string`

`generatePlots` — Display pole-zero map
`off` | `on`

Set to `on` to display pole-zero map for the linear system state-space model. Otherwise, set to `off`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'SourceDocument','MIL1797A'`

`SourceDocument` — Document for flying qualities requirements verification
`MIL8785C` (default) | `MIL1797A`

Document for flying qualities requirements verification, specified as:

- `MIL8785C` — Flying qualities of piloted airlines
- `MIL1797A` — Flying qualities of piloted aircraft

Data Types: `char` | `string`

`Level` — Flying qualities level
`Lowest` (default) | `All` | `1` | `2` | `3`

Flying qualities level, specified as:

- `Lowest` — Returns the verified requirements closest to level 1 for each requirement in the selected source document.

- `All` — Returns a `struct vector` with all requirement levels and their verification status.
- `1`, `2`, or `3` — Returns the desired requirement level, regardless of the verification status.

Data Types: `char` | `string`

Output Arguments

`lonFQOut` — Phugoid and short-period longitudinal flying qualities
structure vector

Phugoid and short-period longitudinal flying qualities, returned as a structure vector.

`varNameOut` — Output results structure
scalar string | `''`

If a linear system is selected through the input dialog, `varNameOut` returns the results structure variable name. Otherwise, `varNameOut` returns an empty string.

Limitations

This function requires the Simulink Control Design license.

Version History

Introduced in R2018b

See Also

`asbFlightControlAnalysis` | `computeLateralDirectionalFlyingQualities` | `linearizeAirframe` | `trimAirframe`

Topics

"Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles" on page 2-65

sim3d.Editor

Interface to the Unreal Engine project

Description

Use the `sim3d.Editor` class to interface with the Unreal Editor.

Note `sim3d.Editor` requires Simulink 3D Animation.

To develop scenes with the Unreal Editor and co-simulate with Simulink, you need the support package. The support package contains an Unreal Engine project that allows you to customize the scenes. For information about the support package, see “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2.

Creation

Syntax

`sim3d.Editor(project)`

Description

MATLAB creates an `sim3d.Editor` object for the Unreal Editor project specified in `sim3d.Editor( project)`.

Input Arguments

`project` — Project path and name
string scalar

Project path and name.

Note Non-ASCII space characters are not supported in the project path.

Example: `"C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"`

Data Types: `string`

Properties

`Uproject` — Project path and name
string scalar

This property is read-only.

Project path and name with Unreal Engine project file extension.

Example: `"C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"`

Data Types: `string`

Object Functions

open   Open the Unreal Editor

Examples

Open Project in Unreal Editor

Open an Unreal Engine project in the Unreal Editor.

Create an instance of the `sim3d.Editor` class for the Unreal Engine project located in `C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject`.

```
editor = sim3d.Editor(fullfile("C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"))
```

Open the project in the Unreal Editor.

```
editor.open();
```

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

`sim3d.Editor` requires Simulink 3D Animation.

See Also

Topics

"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2

linearizeAirframe

Linearize airframe model around operating points

Syntax

```
linSys = linearizeAirframe(modelToAnalyze)
linSys = linearizeAirframe(modelToAnalyze)
linSys = linearizeAirframe(modelToAnalyze,opPoint)
linSys = linearizeAirframe(modelToAnalyze,opPoint,generatePlots)
```

Description

`linSys = linearizeAirframe(modelToAnalyze)` linearizes an airframe model around a specified operating point or operating point specification object and generates an output state-space model that contains only longitudinal states. A **Linearize Airframe** dialog window prompts you to select an operating point or operating point specification object from the base workspace. If an operating point or operating point specification object does not exist in the base workspace, click the **Launch Trim Tool** button in the **Linearize Airframe** dialog window. This button starts the Simulink Control Design Model Linearizer in which you can create the operating point specification object. The `linearizeAirframe` function uses this object as the operating condition around which to linearize the airframe model.

`linSys = linearizeAirframe(modelToAnalyze)` linearizes an airframe model around the specified operating point object or operating point specification object.

`linSys = linearizeAirframe(modelToAnalyze,opPoint)` linearizes an airframe model around the specified operating point object or operating point specification object.

`linSys = linearizeAirframe(modelToAnalyze,opPoint,generatePlots)` displays bode and step plot results of longitudinal linearization.

Examples

Linearize Model Around a Provided Operating Point Specification Object

Linearize the model `SkyHoggAnalysisModel` around the operating point, `opTrim`. This example starts the flight control analysis template using `asbFlightControlAnalysis` and trims the model around the `opSpecDefault` operating point specification object. It then linearizes the airframe model around the `opTrim` operating point and calculates the short- and long-period (phugoid) mode characteristics of `linSys`.

```
asbFlightControlAnalysis('3DOF', 'SkyHoggAnalysisModel');
opSpecDefault = SkyHogg3DOFOpSpec('SkyHoggAnalysisModel');
opTrim = trimAirframe('SkyHoggAnalysisModel', opSpecDefault);
```

```
linSys = linearizeAirframe('SkyHoggAnalysisModel', opTrim)
flyingQual = computeLongitudinalFlyingQualities('SkyHoggAnalysisModel', linSys)
```

Input Arguments

`modelToAnalyze` — Model on which to perform flight control analysis
model name

Model on which to perform flight control analysis. This model must be previously created with the `asbFlightControlAnalysis` function.

Data Types: `char` | `string`

`opPoint` — Operating point object
operating point object

Operating point object used to linearize the model `modelToAnalyze`.

Data Types: `char` | `string`

`generatePlots` — Display pole-zero map
model name

Display pole-zero map for the linear system state-space model.

Data Types: `char` | `string`

Output Arguments

`linSys` — State-space model object
linear state-space model object name

State space model object representing the linearized airframe model at a specified operating point.

Data Types: `char` | `string`

Limitations

This function requires the Simulink Control Design license.

Version History

Introduced in R2018b

See Also

`asbFlightControlAnalysis` | `computeLateralDirectionalFlyingQualities` | `computeLongitudinalFlyingQualities` | `trimAirframe` | Model Linearizer

Topics

"Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles" on page 2-65

linearizeLongitudinalAirframe

Linearize airframe model around operating points

Note This function is not recommended. Use `linearizeAirframe` instead.

Syntax

```
linearizeLongitudinalAirframe(modelToAnalyze)
linearizeLongitudinalAirframe(modelToAnalyze,opPoint)
linearizeLongitudinalAirframe(modelToAnalyze,opPoint,generatePlots)
```

Description

`linearizeLongitudinalAirframe(modelToAnalyze)` linearizes an airframe model around a specified operating point or operating point specification object and generates an output state-space model that contains only longitudinal states. A **Linearize Airframe** dialog window prompts you to select an operating point or operating point specification object from the base workspace. If an operating point or operating point specification object does not exist in the base workspace, click the **Launch Trim Tool** button in the **Trim Airframe** dialog window. This button starts the Simulink Control Design Model Linearizer in which you can create the operating point specification object. From this object, the `linearizeLongitudinalAirframe` function creates the operating point.

`linearizeLongitudinalAirframe(modelToAnalyze,opPoint)` linearizes an airframe model around the specified operating point object or operating point specification object.

`linearizeLongitudinalAirframe(modelToAnalyze,opPoint,generatePlots)` displays bode and step plot results of longitudinal linearization.

Examples

Linearize Model While Specifying an Operating Point Specification Object

Linearize the model `SkyHoggAnalysisModel` and specify an operating point, `opTrim`. This example starts the flight control analysis template using `asbFlightControlAnalysis` and trims the model around the `opSpecDefault` operating point specification object. It then linearizes the airframe model around the `opTrim` operating point and calculates the short- and long-period (phugoid) mode characteristics of `linSys`.

```
asbFlightControlAnalysis('3DOF', 'SkyHoggAnalysisModel');
opSpecDefault = SkyHogg3DOFOpSpec('SkyHoggAnalysisModel');
opTrim = trimAirframe('SkyHoggAnalysisModel', opSpecDefault);
linSys = linearizeLongitudinalAirframe('SkyHoggAnalysisModel', opTrim)
flyingQual = computeLongitudinalFlyingQualities('SkyHoggAnalysisModel', linSys)
```

Input Arguments

`modelToAnalyze` — Model on which to perform flight control analysis
model name

Model on which to perform flight control analysis using the linear state-space model `linSys`. This model must be previously created with the `asbFlightControlAnalysis` function.

Data Types: `char` | `string`

`opPoint` — Linear state-space model
linear state-space model name

Linear state-space model used to perform flight control analysis on `modelToAnalyze`.

Data Types: `char` | `string`

`generatePlots` — Display pole-zero map
model name

Display pole-zero map for the linear system state-space model.

Data Types: `char` | `string`

Limitations

This function requires the Simulink Control Design license.

Version History

Introduced in R2018b

R2019a: `linearizeLongitudinalAirframe` not recommended
Behavior changed in R2019a

This function is not recommended. Use `linearizeAirframe` instead.

See Also

`linearizeAirframe` | `asbFlightControlAnalysis` |
`computeLongitudinalFlyingQualities` | `trimAirframe` | Model Linearizer

Topics

"Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles" on page 2-65

open

Open the Unreal Editor

Syntax

```
[status,result] = open(sim3dEditorObj)
```

Description

Note `open` requires Simulink 3D Animation.

`[status,result] = open(sim3dEditorObj)` opens the Unreal Engine project in the Unreal Editor.

To develop scenes with the Unreal Editor and co-simulate with Simulink, you need the support package. The support package contains an Unreal Engine project that allows you to customize the scenes. For information about the support package, see "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2.

Input Arguments

`sim3dEditorObj` — sim3d.Editor object

`sim3d.Editor` object

`sim3d.Editor` object for the Unreal Engine project.

Output Arguments

`status` — Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status when `command` starts
- If `command` does not include the ampersand character (&), then `status` is the exit status upon `command` completion.

`result` — Output of operating system command

character vector

Output of the operating system command, returned as a character vector. The system shell might not properly represent non-Unicode® characters.

Version History

Introduced in R2021b

R2024a: Requires Simulink 3D Animation

`open` requires Simulink 3D Animation.

See Also

`sim3d.Editor`

ReadSimulation3DMessage

Receives message from Simulink model using a message reader object

Syntax

`status=ReadSimulation3DMessage(MessageReader, dataSize, data)`

Description

`status=ReadSimulation3DMessage(MessageReader, dataSize, data)` receives a message from a Simulink model using a message reader object.

The C++ syntax is

```
int ReadSimulation3DMessage(void *MessageReader, uint32 dataSize, void *data);
```

Input Arguments

`MessageReader` — Pointer to message reader object
object pointer

Pointer to message reader object, `ReadSimulation3DMessage`.

Data Types: `void *`

`dataSize` — Size of data
`number of bytes` | `scalar`

Size of data, that is, `data (sizeof(datatype) *num_of_elements)`. For example, if you want to read a vector of 3 floats, the data size is `sizeof(float)*3`.

Data Types: `uint32`

`data` — Pointer to data object
object pointer

Pointer to data object.

Data Types: `void *`

Output Arguments

`status` — Operation exit status
0 | nonzero integer

Status, returned as either `0` or a nonzero integer. When the operation is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

Version History

Introduced in R2021b

See Also
`ASim3dActor`

External Websites
Unreal Engine Documentation

Sim3dRelease

C++ method that releases actor in Unreal Engine 3D simulation

Syntax

`void ASetGetActorLocation::Sim3dRelease()`

Description

The C++ method `void ASetGetActorLocation::Sim3dRelease()` releases an actor in the Unreal Engine 3D simulation environment. The Unreal Engine `AActor::EndPlay` class calls the `Sim3dRelease` method when the 3D simulation ends.

Examples

Release Actor

```
void ASetGetActorLocation::Sim3dRelease()
{
   Super::Sim3dRelease();
   if (MessageReader) {
          StopSimulation3DMessageReader (SignalReader);
   }
   MessageReader = nullptr;

   if (MessageWriter) {
          StopSimulation3DMessageWriter (SignalWriter);
   }
   MessageWriter = nullptr;
}
```

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

Sim3dSetup

C++ method that sets up actor in Unreal Engine 3D simulation

Syntax

`void ASetGetActorLocation::Sim3dSetup()`

Description

The C++ method `void ASetGetActorLocation::Sim3dSetup()` sets up an actor in the Unreal Engine 3D simulation environment. The Unreal Engine `AActor::BeginPlay` class calls the `Sim3dSetup` method every frame.

Examples

Set Up Actor

```
void ASetGetActorLocation::Sim3dSetup()
{
	Super::Sim3dSetup();
		if (Tags.Num() != 0) {
		    FString tagName = Tags.Top().ToString();

		    FString MessageReaderTag = tagName;
		    MessageReaderTag.Append(TEXT("SimulinkMessage_OUT")); // a message from Simulink model
		    MessageReader = StartSimulation3DMessageReader (TCHAR_TO_ANSI(*MessageReaderTag), MAX_MESSAGE_SIZE);

		    FString MessageWriterTag = tagName;
		    MessageWriterTag.Append(TEXT("SimulinkMessage_IN")); // a message to Simulink model
		    MessageWriter = StartSimulation3DMessageWriter (TCHAR_TO_ANSI(*MessageWriterTag) ), MAX_MESSAGE_SIZE);
		}
}
```

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

Sim3dStep

C++ method that steps actor in Unreal Engine 3D simulation

Syntax

`void ASetGetActorLocation::Sim3dStep(float DeltaSeconds)`

Description

The C++ method `void ASetGetActorLocation::Sim3dStep(float DeltaSeconds)` steps an actor in the Unreal Engine 3D simulation environment. The Unreal Engine `AActor::Tick` class calls the `Sim3dStep` method.

Examples

Step Actor

```
void ASetGetActorLocation::Sim3dStep(float DeltaSeconds)
{
	Super::Sim3dStep(DeltaSeconds);
	uint32 messageSize = MAX_MESSAGE_SIZE;
	int statusR = ReadSimulation3DMessage (MessageReader, &messageSize, message);
	...
	int statusW = WriteSimulation3DMessage (MessageWriter, messageSize, message);
}
```

Input Arguments

`DeltaSeconds` — Elapsed time
`.01`

Time elapsed since Unreal Engine modified the frame.

Data Types: `float`

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

StartSimulation3DMessageReader

Constructs a message reader object in the Unreal Editor

Syntax

`MessageReader = StartSimulation3DMessageReader(topicName, maxDataSize)`

Description

`MessageReader = StartSimulation3DMessageReader(topicName, maxDataSize)` constructs a message reader object in the Unreal Editor.

The C++ syntax is

```
void *StartSimulation3DMessageReader(const char* topicName,uint32 maxDataSize);
```

Input Arguments

`topicName` — Simulink signal topic name
`mySignal`

Name of the Simulink signal with the message topic.

Data Types: `char *`

`maxDataSize` — Maximum size of data
`number of bytes` | `scalar`

Maximum size of the data, in bytes.

Data Types: `uint32`

Output Arguments

`MessageReader` — Pointer to message reader object
object pointer

Pointer to message reader object, `ReadSimulation3DMessage`.

Data Types: `void *`

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

StartSimulation3DMessageWriter

Constructs a message writer object in the Unreal Editor

Syntax

```
MessageWriter = StartSimulation3DMessageWriter(topicName, maxDataSize)
```

Description

`MessageWriter = StartSimulation3DMessageWriter(topicName, maxDataSize)` constructs a message writer object in the Unreal Editor.

The C++ syntax is

```
void *StartSimulation3DMessageWriter(const char* topicName, uint32 maxDataSize);
```

Input Arguments

`topicName` — Simulink signal topic name
`mySignal`

Name of the Simulink signal with the message topic.

Data Types: `char *`

`maxDataSize` — Maximum size of data
`number of bytes` | `scalar`

Maximum size of the data, in bytes.

Data Types: `uint32`

Output Arguments

`MessageWriter` — Pointer to message writer object
object pointer

Pointer to message writer object, `WriteSimulation3DMessage`.

Data Types: `void *`

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

StopSimulation3DMessageReader

Deletes message reader object in the Unreal Editor

Syntax

`status=StopSimulation3DMessageReader(MessageReader)`

Description

`status=StopSimulation3DMessageReader(MessageReader)` deletes the Unreal Editor 3D message reader object.

The C++ syntax is

```
int StopSimulation3DMessageReader(void * MessageReader);
```

Input Arguments

`MessageReader` — Pointer to message reader object
object pointer

Pointer to message reader object, `ReadSimulation3DMessage`.

Data Types: `void *`

Output Arguments

`status` — Operation exit status
0 | nonzero integer

Status, returned as either `0` or a nonzero integer. When the operation is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

StopSimulation3DMessageWriter

Deletes message writer object in the Unreal Editor

Syntax

`status=StopSimulation3DMessageWriter(MessageWriter)`

Description

`status=StopSimulation3DMessageWriter(MessageWriter)` deletes the Unreal Editor 3D message writer object.

The C++ syntax is

```
int StopSimulation3DMessageWriter(void *MessageWriter);
```

Input Arguments

`MessageWriter` — Pointer to message writer object
object pointer

Pointer to message writer object, `WriteSimulation3DMessage`.

Data Types: `void *`

Output Arguments

`status` — Operation exit status
0 | nonzero integer

Status, returned as either `0` or a nonzero integer. When the operation is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

trimAirframe

Trim airframe around operating point specification object

Syntax

```
trimAirframe(modelToAnalyze)
trimAirframe(modelToAnalyze,opSpec)
```

Description

`trimAirframe(modelToAnalyze)` trims the airframe around an operating point specification object. A **Trim Airframe** dialog window prompts you to select an operating point specification object from the base workspace. If an operating point specification object does not exist in the base workspace, click the **Launch Trim Tool** button in the **Trim Airframe** dialog window. This button starts the Simulink Control Design Model Linearizer in which you can create the operating point specification object. From this object, the `trimAirframe` function trims the airframe.

`trimAirframe(modelToAnalyze,opSpec)` trims the airframe model around the specified operating point specification object.

Examples

Trim Model While Specifying an Operating Point Specification Object

Trim the model `SkyHoggAnalysisModel` around an operating point specification object, `opSpecDefault`. This example starts the flight control analysis template using `asbFlightControlAnalysis` and trims the model around the `opSpecDefault` operating point. It then linearizes the airframe model around the `opTrim` operating point specification object and calculates the short- and long-period (phugoid) mode characteristics of `linSys`.

```
asbFlightControlAnalysis('3DOF', 'SkyHoggAnalysisModel');
opSpecDefault = SkyHogg3DOFOpSpec('SkyHoggAnalysisModel');
opTrim = trimAirframe('SkyHoggAnalysisModel', opSpecDefault);
linSys = linearizeAirframe('SkyHoggAnalysisModel', opTrim)
flyingQual = computeLongitudinalFlyingQualities('SkyHoggAnalysisModel', linSys)
```

Input Arguments

`modelToAnalyze` — Model on which to perform flight control analysis
model name

Model on which to perform flight control analysis using the linear state-space model `linSys`. This model must be previously created with the `asbFlightControlAnalysis` function.

Data Types: `char` | `string`

`opSpec` — Linear state-space model
linear state-space model name

Linear state-space model used to perform flight control analysis on `modelToAnalyze`.

Data Types: `char` | `string`

Limitations

This function requires the Simulink Control Design license.

Version History

Introduced in R2018b

See Also

`asbFlightControlAnalysis` | `computeLateralDirectionalFlyingQualities` | `computeLongitudinalFlyingQualities` | `linearizeAirframe` | Model Linearizer

Topics

"Analyze Dynamic Response and Flying Qualities of Aerospace Vehicles" on page 2-65

WriteSimulation3DMessage

Sends message to Simulink model using a message writer object

Syntax

`status=WriteSimulation3DMessage(MessageWriter, dataSize, data)`

Description

`status=WriteSimulation3DMessage(MessageWriter, dataSize, data)` sends a message to a Simulink model using a message writer object.

The C++ syntax is

```
int WriteSimulation3DMessage(void * MessageWriter, uint32 dataSize, void *data);
```

Input Arguments

`MessageWriter` — Pointer to message writer object
object pointer

Pointer to message writer object, `WriteSimulation3DMessage`.

Data Types: `void *`

`dataSize` — Size of data
`number of bytes` | `scalar`

Size of data, that is, `data (sizeof(datatype) *num_of_elements)`. For example, if you want to read a vector of 3 floats, the data size is `sizeof(float)*3`.

Data Types: `uint32`

`data` — Pointer to data object
object pointer

Pointer to data object.

Data Types: `void *`

Output Arguments

`status` — Operation exit status
0 | nonzero integer

Status, returned as either `0` or a nonzero integer. When the operation is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

Version History

Introduced in R2021b

See Also

`ASim3dActor`

External Websites

Unreal Engine Documentation

sim3d.maps

Access additional scenes from the server

Description

Use the `sim3d.maps` to download and access additional scenes from the server so that they can be automatically available in the Simulation 3D Scene Configuration block.

Object Functions

sim3d.maps.Map.download	Download maps from the server
sim3d.maps.Map.server	List of maps available for download from the server
sim3d.maps.Map.delete	Delete local maps downloaded from the server
sim3d.maps.Map.local	List of locally available maps

Troubleshooting

- If you cannot reach the server, the download will fail due to a timeout.
- If the download fails while updating an existing map, the existing outdated file will remain functional.
- If you delete the CSV file, you will lose automatic tracking of updates for the existing maps.

Version History

Introduced in R2022b

See Also

Simulation 3D Scene Configuration

sim3d.maps.Map.delete

Delete local maps downloaded from the server

Syntax

```
sim3d.maps.Map.delete(Scene)
```

Description

`sim3d.maps.Map.delete(Scene)` deletes the map `Scene` from your local system.

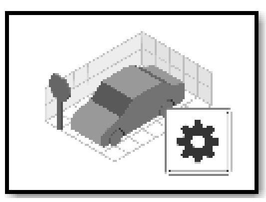
Examples

Download Suburban Scene Map

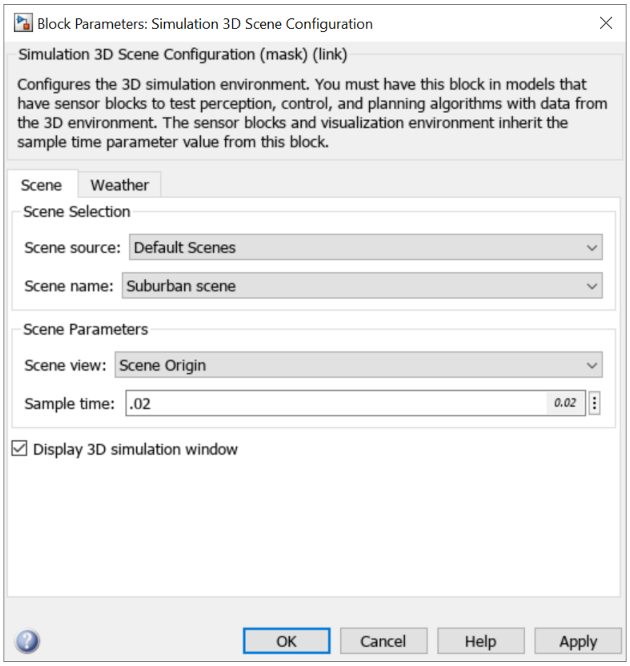
This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.



Run the model.

Delete the model and check if the map is till available locally.

```
sim3d.maps.Map.delete('Suburban scene')
```

```
Suburban scene was successfully deleted
```

Input Arguments

Scene — Name of scene

string | character array

Name of the map being deleted, specified as a string or character array. Once the map is deleted, it automatically disappears from the Simulation 3D Scene Configuration block mask menu.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.download` | `sim3d.maps.Map.server` | `sim3d.maps.Map.local`

sim3d.maps.Map.download

Download maps from the server

Syntax

```
sim3d.maps.Map.download(Scene)
```

Description

`sim3d.maps.Map.download(Scene)` downloads the map `Scene` from the server.

Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

```
        MapName                      Description                     Version
    ________________    __________________________________________    _______

    "Suburban scene"    "a suburban area beyond the city's border"      "1"
```

Download the Suburban scene from the server.

```
sim3d.maps.Map.download('Suburban scene')
```

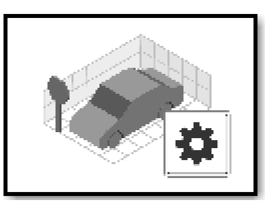
```
Map is susccesfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

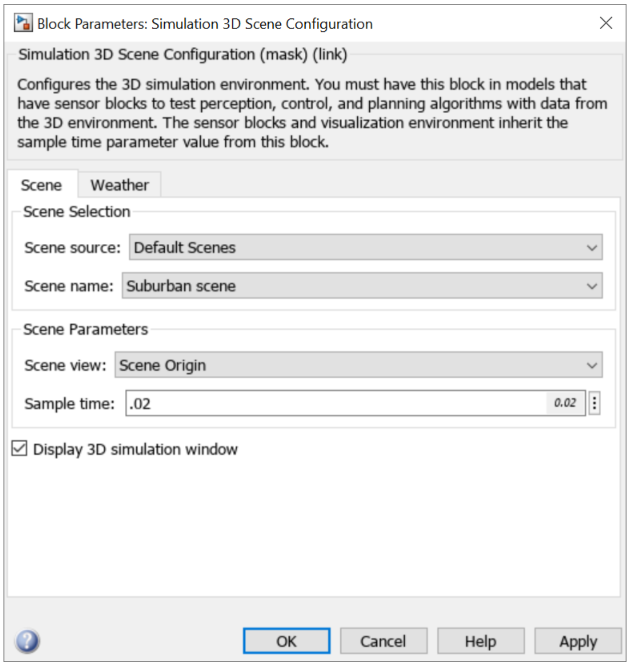
```
sim3d.maps.Map.local
```

```
        MapName                      Description                     Version
    ________________    __________________________________________    _______

    "Suburban scene"    "a suburban area beyond the city's border"      "1"
```

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.



Run the model.

Input Arguments

Scene — Name of scene

string | character array

Name of the map being downloaded from the server, specified as a string or character array. Maps are downloaded in the default folder that is added to MATLAB search path at startup.

Maps are stored by user profile. For multiuser setup with a single MATLAB installation, the maps will be downloaded multiple times.

If a new version of the map is available on the server, you will see a warning message asking you to download the map again to get the recent version.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.server` | `sim3d.maps.Map.delete` | `sim3d.maps.Map.local`

sim3d.maps.Map.local

List of locally available maps

Syntax

```
sim3d.maps.Map.local
```

Description

`sim3d.maps.Map.local` lists the locally available maps.

Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

```
        MapName                        Description                         Version
    ________________    __________________________________________    _______

    "Suburban scene"    "a suburban area beyond the city's border"      "1"
```

Download the Suburban scene from the server.

```
sim3d.maps.Map.download('Suburban scene')
```

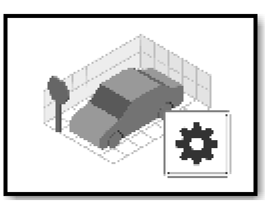
```
Map is susccesfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

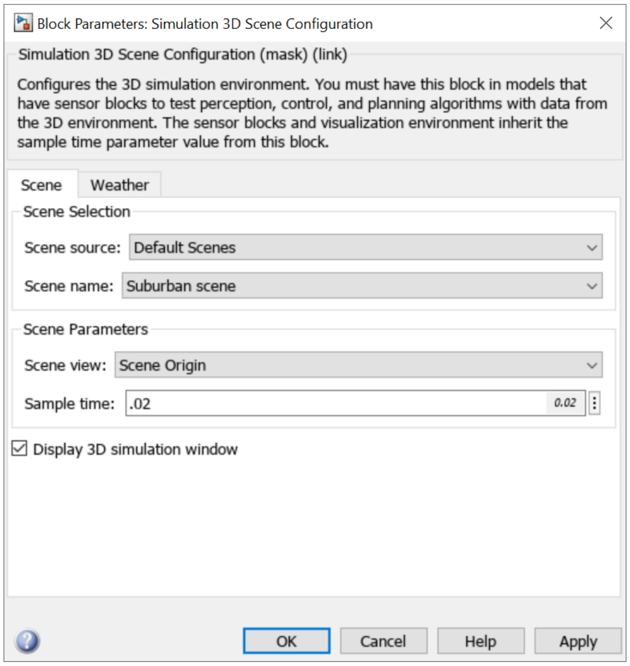
```
sim3d.maps.Map.local
```

```
        MapName                        Description                         Version
    ________________    __________________________________________    _______

    "Suburban scene"    "a suburban area beyond the city's border"      "1"
```

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.



Run the model.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.download` | `sim3d.maps.Map.server` | `sim3d.maps.Map.delete`

sim3d.maps.Map.server

List of maps available for download from the server

Syntax

`sim3d.maps.Map.server`

Description

`sim3d.maps.Map.server` lists the available maps in the server.

Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

```
        MapName                      Description                     Version
    ________________    __________________________________________    _______

    "Suburban scene"    "a suburban area beyond the city's border"      "1"
```

Download the Suburban scene from the server.

```
sim3d.maps.Map.download('Suburban scene')
```

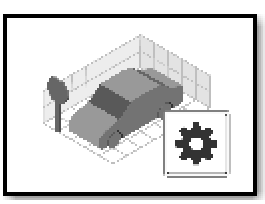
```
Map is succcesfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

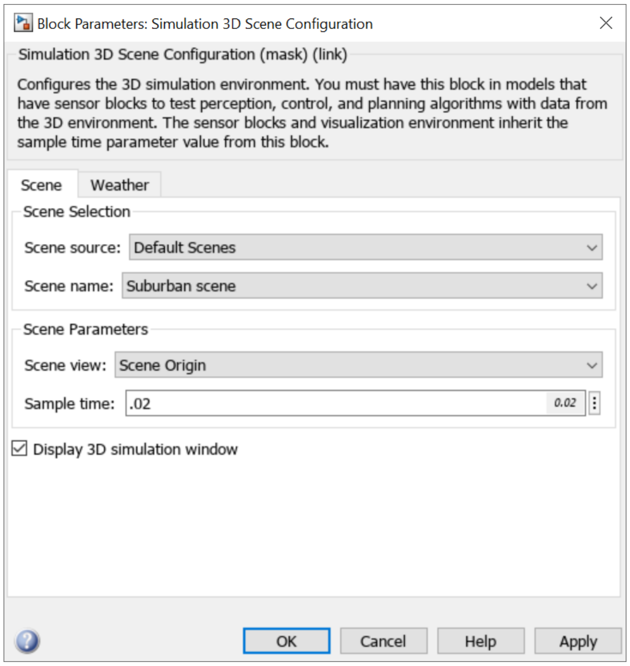
```
sim3d.maps.Map.local
```

```
        MapName                      Description                     Version
    ________________    __________________________________________    _______

    "Suburban scene"    "a suburban area beyond the city's border"      "1"
```

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.



Run the model.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.download` | `sim3d.maps.Map.delete` | `sim3d.maps.Map.local`

copyExampleSim3dProject

Copy support package files and plugins to specified folders

Syntax

```
sim3d.utils.copyExampleSim3dProject(DestFldr)
sim3d.utils.copyExampleSim3dProject(DestFldr,Name=Value)
```

Description

`sim3d.utils.copyExampleSim3dProject(DestFldr)` copies the Aerospace Blockset Interface for Unreal Engine Projects support package project files to the destination folder, `DestFldr`. By default, `copyExampleSim3dProject` copies the plugins to your Epic Games installation folder.

`sim3d.utils.copyExampleSim3dProject(DestFldr,Name=Value)` copies support package files to the destination with additional options specified by name-value arguments. If your Epic Games installation is not saved to the default location `C:\Program Files\Epic Games`, specify the `PluginDestination` argument.

Running the `sim3d.utils.copyExampleSim3dProject` function configures your environment so that you can customize scenes. The support package contains these Aerospace Blockset Interface for Unreal Engine Projects components.

- An Unreal project, defined in `AutoVrtlEnv.uproject`, and its associated files. The project includes editable versions of the prebuilt 3D scenes that you can select from the **Scene name** parameter of the Simulation 3D Scene Configuration block.
- Three plugins: `MathWorkSimulation`, `RoadRunnerMaterials`, and `MathWorksAerospaceContent`. These plugins establish the connection between MATLAB and the Unreal Editor and are required for co-simulation.

Input Arguments

`DestFldr` — Destination folder for Unreal project files

character vector

Destination folder name, specified as a character vector.

Running `copyExampleSim3dProject` copies the Unreal project, defined in `AutoVrtlEnv.uproject`, and its associated project files to the destination folder.

If the `DestFldr` argument is specified as `""`, the `copyExampleSim3dProject` function does not copy the project files.

Note You must have write permission for the destination folder.

Example: `C:\project`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

`Source` — Support package source folder
character vector

Support package source folder, specified as a character vector. The folder contains the downloaded support packages files.

By default, if you do not specify the source folder, `copyExampleSim3dProject` copies the file from the support package installation folder, `matlabshared.supportpkg.getSupportPackageRoot()`.

Example: `Source="shared\sim3dprojects\spkg\"`

Data Types: `char` | `string`

`PluginDestination` — Option to change the plugin destination folder
character vector

Option to change the plugin destination folder, specified as a character vector.

By default, if you do not change the plugin installation folder location, `copyExampleSim3dProject` tries to copy the plugins to `C:\Program Files\Epic Games\UE_5.1\Engine\Plugins \Marketplace\MathWorks`.

Example: `PluginDestination="C:\Program Files\Epic Games\UE_5.1\Engine\Plugins \Marketplace\MathWorks"`

Data Types: `char` | `string`

`VerboseOutput` — Option to enable verbose logging
`0` or `false` (default) | `1` or `true`

Option to enable verbose logging, specified as a logical `0` (false) or `1` (true). Verbose logging displays intermediate iteration information on the MATLAB command line.

Example: `VerboseOutput=true`

Data Types: `logical`

`SkipPluginCopy` — Option to skip copying plugins to the Unreal Engine directory
`false` (default) | `true`

Option to skip copying plugins to the Unreal Engine directory, specified as a logical `false` or `true`.

Note Use this option if you previously copied plugins to the Unreal Engine directory.

Example: `PluginDestination=true`

Data Types: `logical`

Examples

Copy Support Package Files to Destination Folder

Copy the support package files to `C:\project`.

```
sim3d.utils.copyExampleSim3dProject("C:\project");
```

Copy the support package files to `C:\project` with `VerboseOutput` set to `true`.

```
sim3d.utils.copyExampleSim3dProject("C:\project", VerboseOutput=true)
```

```
Copying ...\spkg\project\AutoVrtlEnv to C:\project\AutoVrtlEnv
Creating C:\project\AutoVrtlEnv\Plugins
Copying ...\spkg\plugins\mw_aerospace\MathWorksAerospace to C:\project\AutoVrtlEnv\Plugins\MathWo
Copying ...\spkg\plugins\mw_automotive\MathWorksAutomotiveContent to C:\project\AutoVrtlEnv\Plugi
Copying ...\spkg\plugins\mw_simulation\MathWorksSimulation to C:\project\AutoVrtlEnv\Plugins\Math
Copying ...\spkg\plugins\mw_uav\MathWorksUAVContent to C:\project\AutoVrtlEnv\Plugins\MathWorksU
Copying ...\spkg\plugins\rr_materials\RoadRunnerMaterials to C:\project\AutoVrtlEnv\Plugins\RoadR
Ensuring C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject is writable
Enabling plugin MathWorksSimulation in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
Enabling plugin MathWorksUAVContent in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
Enabling plugin MathWorksAutomotiveContent in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
Enabling plugin RoadRunnerMaterials in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
```

Version History

Introduced in R2022b

See Also

Topics

"Install Support Package and Configure Environment" on page 4-3
"How 3D Simulation for Aerospace Blockset Works" on page 2-40
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37

External Websites

Unreal Engine
Using Unreal Engine with Simulink

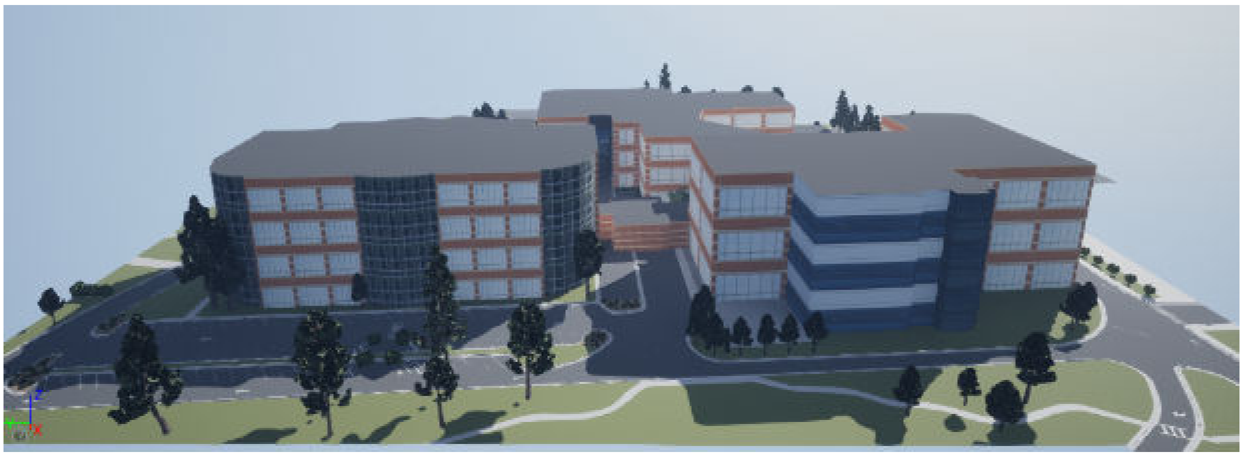
Scenes

Apple Hill

Apple Hill campus in Natick, MA

Description

The Apple Hill scene is a 3D environment of the MathWorks Apple Hill campus in Natick, Massachusetts. The scene is rendered with Unreal Engine.



Setup

This scene is available only in the Aerospace Blockset Interface for Unreal Engine Projects support package. After installing this support package, follow these steps to use the scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In the Block Parameters dialog box, set the **Scene source** parameter to `Unreal Editor`.
3. Set the **Project** parameter to your project file.
4. Click the **Open Unreal Editor** button.
5. In the Unreal Editor, load the Apple Hill map by double-clicking **MathWorksAerospaceContent Content > Maps > AppleHill**.

Layout

The Apple Hill scene coordinate directions align with positive *X* pointing North and positive *Y* pointing East. Blocks in the Aerospace Simulation 3D library require dimensions in meters. For convenience, this section also provides dimensions in feet.

Scene Dimensions

Overall Dimensions	Length	Width
Feet	726.0	862.9
Meters	221.3	263.0

The entire scene dimension is [-152.8,-167.6] to [68.5,95.4] m.

Tips

- The Aerospace Blockset Interface for Unreal Engine Projects support package is required to use this scene. You can also modify this scene.

  For more details on customizing scenes, see “Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2.

Version History

Introduced in R2023b

See Also

Blocks
Simulation 3D Scene Configuration

Tools
Airport | Griffiss Airport

Topics
“Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37
“Customize 3D Scenes for Aerospace Blockset Simulations” on page 4-2
“How 3D Simulation for Aerospace Blockset Works” on page 2-40

Airport

Generic airport

Description

The Airport scene is a 3D environment of a generic airport. The scene is rendered with Unreal Engine.

Setup

To use this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In the Block Parameters dialog box, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `Airport`.

Layout

Blocks in the Aerospace Simulation 3D library require dimensions in meters. For convenience, this section also provides dimensions in feet.

Scene Dimensions

This table provides the runway dimensions.

Overall Dimensions	Length	Width
Feet	9900	150
Meters	3318	46

The entire scene dimension is [0, 0, 0] to [10082.5, 0, 0] m.

Taxiway Width

This table provides the taxiway widths.

Taxiway Width	Width
Feet	100
Meters	30

Elevation

The elevation of the airport runway is 0.01 meters.

Runway End Coordinates

The runway end coordinates at the centerline are [3532.49, 0, 0.01] to [6550.01, 0, 0.01] m.

Tips

- If you have the Aerospace Blockset Interface for Unreal Engine Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `Airport`.

  For more details on customizing scenes, see "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2.

Version History

Introduced in R2021b

See Also

Blocks

Simulation 3D Scene Configuration

Tools

Apple Hill | Griffiss Airport

Topics

"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2

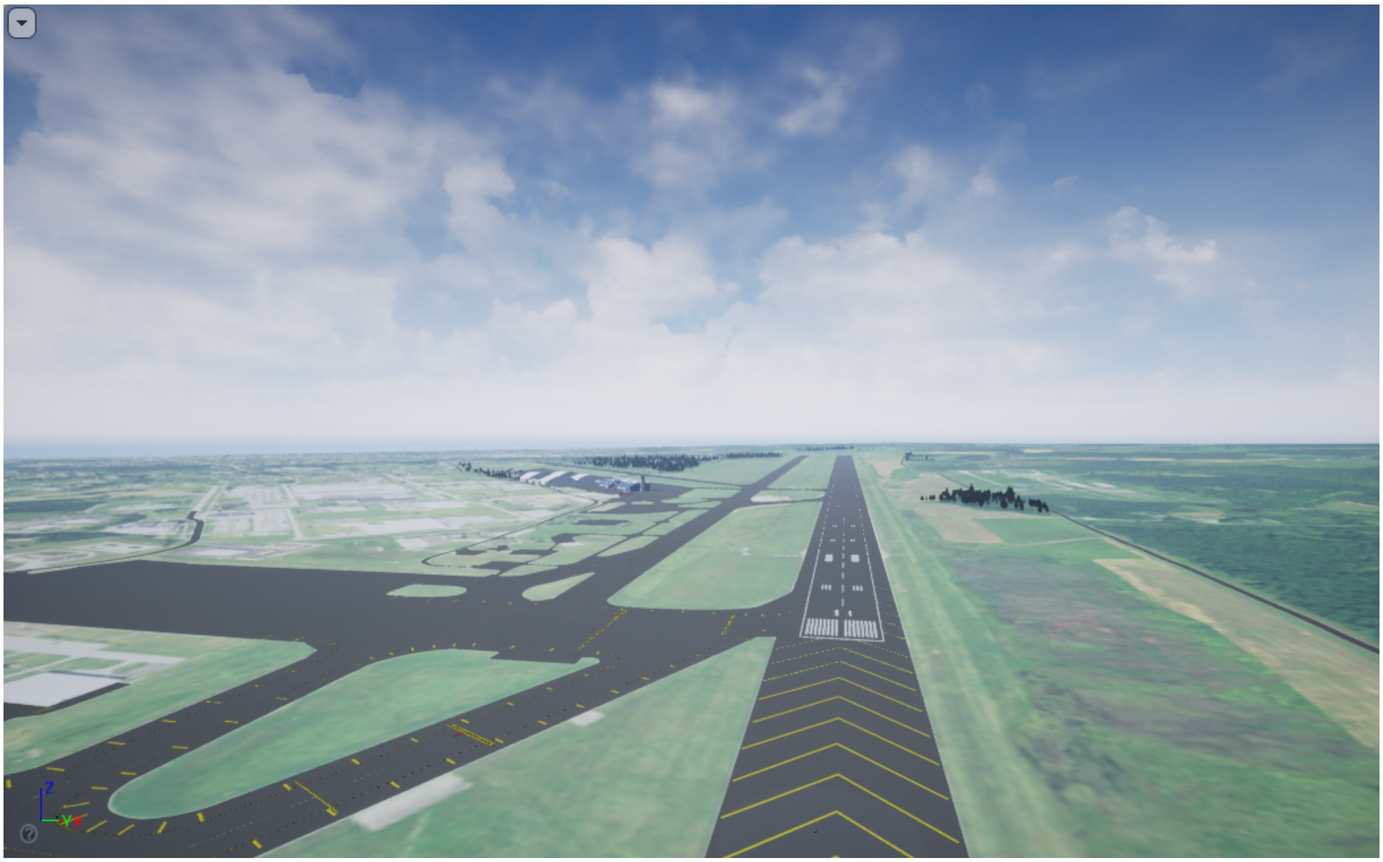
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Griffiss Airport

Griffiss International Airport in Rome, NY

Description

The Griffiss Airport scene is a 3D environment of the Griffiss Airport in Rome, New York. The scene is rendered with Unreal Engine.



Setup

This scene is available only in the Aerospace Blockset Interface for Unreal Engine Projects support package. After installing this support package, follow these steps to use the scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In the Block Parameters dialog box, set the **Scene source** parameter to `Unreal Editor`.
3. Set the **Project** parameter to your project file.
4. Click the **Open Unreal Editor** button.
5. In the Unreal Editor, load the Griffiss Airport map by double-clicking **MathWorksAerospaceContent Content** > **Maps** > **GriffissAirport**.

Layout

The Griffiss Airport scene coordinate directions align with positive *X* pointing North and positive *Y* pointing East. Blocks in the Aerospace Simulation 3D library require dimensions in meters. For convenience, this section also provides dimensions in feet.

Scene Dimensions

This table provides the relevant dimensions for runway 15/33.

Overall Dimensions	Length	Width
Feet	11,820	200
Meters	3603	61

The entire scene dimension is [-2122.57, -5544.71, 132.01] to [5732.00, 2623.05, 323.99] m.

Location

Location	Runway 15	Runway 33
Latitude	43-14.704480N	43-13.351312N
Longitude	075-25.377805W	075-23.465642W
Elevation	504 ft/153.6 m	498 ft/151.8 m
Runway heading	147 degrees magnetic, 134 degrees true	327 degrees magnetic, 314 degrees true
Visual slope indicator		4-light PAPI on left (3.00° glide path)
Approach lights		Medium Intensity Approach Lighting System with Runway Alignment Indicator Lights (MALSR): 1400 ft/426.72 m medium intensity

Runway 33 Touchdown Point

The runway 33 touchdown point is [200.1, -194.7, 147.4] m.

Runway 15 Touchdown Point

The runway 15 touchdown point is [2663.36, -2741.67, 149.52] m.

More About

Lighting

The primary lights of the airport use Niagara particle lighting:

- Runway threshold lights
- Runway edge lights
- Taxiway edge lights

- Precision approach path indicator (PAPI) lights

All Niagara light coding and settings, except those in the level blueprint are located in the **Environment > GriffissAssets > Niagara** folder.

Light	Mesh Object Name in Scene
Runway	`Runway_and_Taxi_Edge_Lights_LoD0Node`
Taxiway	`Runway_and_Taxi_Edge_Lights_LoD0Node`
MALSR	`SM_MALSR_Single_LoD0Node`
Runway threshold	None
Precision approach path indicator (PAPI)	None

Runway 33 uses the precision approach path indicator (PAPI) lights, which are always on and guide aircraft during landing. The Griffiss Airport map does not contain static mesh models for the PAPI lights to the left of the touchdown point on runway 33.

For more information on how turn on and off the lights for the runway, taxiway, threshold, and MALSR, see “Turn Airport Lights On and Off” on page 4-33.

Runway Threshold Lights

The runway threshold lights are at each end of the runway where the landing area begins. The map embeds these lights in the pavement so they are visible to approaching aircraft at night. The Griffiss Airport map models these lights with Niagara particle lights.

The Griffiss Airport map runway threshold lights change color depending on the direction in which you view them. The lighting color change is coded in the material function `MF_RunwayColorLogic`.

On approach to landing:

- The lights at the threshold at the beginning of the runway are green.
- The lights at the threshold at the end of the runway are red.

PAPI Lights

PAPI lights help pilots keep their aircraft on a recommended glide slope for the runway. The Griffiss Airport map models the PAPI lights using Niagara particle lights. PAPI lights are always on.

The Griffiss Airport map sets the PAPI lights to a glide slope of three degrees.

- When an aircraft is on the glide slope:
  - The two outer lights (farthest from the runway edge) are white.
  - The two inner lights are red.
- When an aircraft is above the glide slope, the three or four outer lights are white.
- When an aircraft is below the glide slope, the three or four inner lights are red.

To set the angle of each of the four lights individually, see the material function `MF_PAPIColorLogic`.

Set the actual light angles, light intensity factors, and other light parameters in these eight material instances:

- `MI_BloomBoost_PAPI1`
- `MI_BloomBoost_PAPI2`
- `MI_BloomBoost_PAPI3`
- `MI_BloomBoost_PAPI4`
- `MI_ParticleBloom_PAPI1`
- `MI_ParticleBloom_PAPI2`
- `MI_ParticleBloom_PAPI3`
- `MI_ParticleBloom_PAPI4`

Tips

- The Aerospace Blockset Interface for Unreal Engine Projects support package is required to use this scene. You can also modify this scene.

  For more details on customizing scenes, see "Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2.

Version History

Introduced in R2022b

See Also

Blocks
Simulation 3D Scene Configuration

Tools
Airport | Apple Hill

Topics
"Griffiss Airport Lighting" on page 4-33
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

External Websites
https://www.airnav.com/airport/KRME

Aircraft

Airliner

Generic airliner

Description

Airliner is one of the aircraft that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. For detailed views of the Airliner, see "Views" on page 8-3.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Aircraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Aircraft Parameters** tab, set the **Type** parameter to `Airliner`.
3. Set the **Initial translation (in meters)** and **Initial rotation (in radians)** parameters to an array size that matches the Airliner aircraft, for example, `zeros(12,3)`.

Landing Gear

The airliner landing gear is retractable. It retracts and extends automatically at a preset height of 100 feet above the terrain. You can change this height in the animation blueprint, located at **MathWorksAerospaceContent Content** > **Vehicles** > **Aircraft** > **MWAirliner** > **Animation** > **MWAirlinerAnimBP**.

Data for Aircraft Placement

The `Airliner` sample mesh origin is near its center of mass, 3.030 meters above the bottoms of the tires. To correctly place the aircraft, consider using these values.

Airport Scene Placement

To place the `Airliner` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, -3.030] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, -0.00021, 0]`

Altitude Sensor

For the altitude sensor in the Simulation 3D Aircraft block, use these values.

Parameter	Value
Body Z offset (m)	3.0299
Front tire radius (m)	0.345
Left tire radius (m)	0.562
Right tire radius (m)	0.562

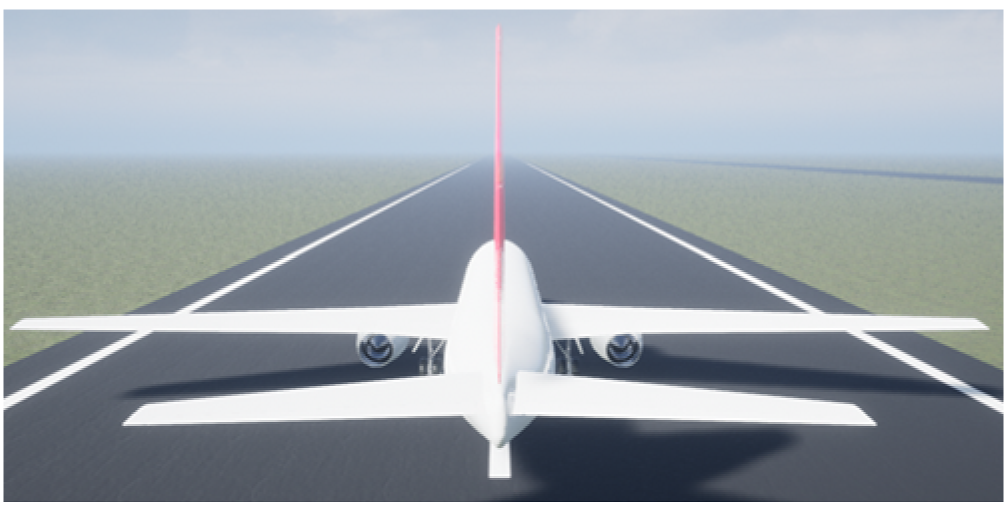
Views

`Top-down view` — Airliner top-down view
diagram

`Side view` — Airliner side view
diagram

`Front view` — Airliner front view
diagram

`Back view` — Airliner back view
diagram



Lights and Skeleton

Lights

Light	Bone	Location in SK_MWAirliner Mesh
Left landing	MWAirliner_Aircraft_Landing_Lights_Wing_L	Left wing root
Right landing	MWAirliner_Aircraft_Landing_Lights_Wing_R	Right wing root
Taxi	MWAirliner_Aircraft_Landing_Lights_F	Nose gear strut
Red navigation	MWAirliner_Aircraft_RedNavigationLight	Left wingtip
Green navigation	MWAirliner_Aircraft_GreenNavigationLight	Right wingtip
Left wingtip strobe	MWAirliner_Aircraft_NavigationLight_L	Left wingtip
Right wingtip strobe	MWAirliner_Aircraft_NavigationLight_R	Right wingtip
Tail strobe	MWAirliner_Aircraft_NavigationLightFin	Top of vertical stabilizer, pointing aft
Beacon #1	MWAirliner_Aircraft_RedAntiCollisionLight1	Top of fuselage
Beacon #2	MWAirliner_Aircraft_RedAntiCollisionLight2	Bottom of fuselage

Skeleton

- `MWAirliner`
  - `MWAirliner_Aircraft_Body_Cap_Left`
    - `MWAirliner_Aircraft_Body_Cap_Element_L`
  - `MWAirliner_Aircraft_Body_Cap_Right`
    - `MWAirliner_Aircraft_Body_Cap_Element_R`
  - `MWAirliner_Aircraft_RedAntiCollisionLight1`
  - `MWAirliner_Aircraft_NavigationLightFin`
  - `MWAirliner_Aircraft_NavigationLight_R`
  - `MWAirliner_Aircraft_GreenNavigationLight`
  - `MWAirliner_Aircraft_NoseWheelStrut_Front`
    - `MWAirliner_Aircraft_Tire_F`
      - `MWAirliner_Aircraft_Wheels_F`
  - `MWAirliner_Aircraft_Body_Elements02`

- MWAirliner_Aircraft_NoseWheelStruct_Left
  - MWAirliner_Aircraft_Tires_L
    - MWAirliner_Aircraft_Wheels_L
  - MWAirliner_Aircraft_NoseWheelStrut_Element_L
- MWAirliner_Aircraft_Elevator_R
- MWAirliner_Aircraft_Body_Fin
- MWAirliner_Aircraft_NoseWheelStrut_Right
  - MWAirliner_Aircraft_NoseWheelStrut_Element_R
  - MWAirliner_Aircraft_Tires_R
    - MWAirliner_Aircraft_Wheels_R
- MWAirliner_Aircraft_RedAntiCollisionLight2
- MWAirliner_Aircraft_JetEngineFan_L
- MWAirliner_Aircraft_JetEngineFan_R
- MWAirliner_Aircraft_Landing_Lights_Wing_L
- MWAirliner_Aircraft_NavigationLight_L
- MWAirliner_Aircraft_Aircraft_Cap_L
- MWAirliner_Aircraft_Aircraft_Cap_R
- MWAirliner_Aircraft_RedNavigationLight
- MWAirliner_Aircraft_Flaps_1_R
- MWAirliner_Aircraft_Ailerons_R
- MWAirliner_Aircraft_JetEngine_R
- MWAirliner_Aircraft_Flaps_2_R
- MWAirliner_Aircraft_Rudder
- MWAirliner_Aircraft_Ailerons_L
- MWAirliner_Aircraft_Flaps_1_L
- MWAirliner_Aircraft_Flaps_2_L
- MWAirliner_Aircraft_Body_Elements01
- MWAirliner_Aircraft_Vertical_Element
- MWAirliner_Aircraft_JetEngine_L
- MWAirliner_Aircraft_Wing_R
- MWAirliner_Aircraft_Landing_Lights_F
- MWAirliner_Aircraft_Body
- MWAirliner_Aircraft_Elevator_L
- MWAirliner_Aircraft_Landing_Lights_Wing_R
- MWAirliner_Aircraft_Glass

Version History

Introduced in R2021b

See Also

Blocks
Simulation 3D Aircraft

Tools
Air Transport | Custom | General Aviation | Sky Hogg

Topics
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Custom

Customizable generic fixed-wing aircraft, including vertical takeoff and landing (VTOL) aircraft

Description

Custom is one of the aircraft that you can use within the 3D simulation environment. It is the same skeleton as that used for the HL-20 aircraft. This environment is rendered using the Unreal Engine from Epic Games. This aircraft is designed to be customized with a user-supplied skeletal mesh. Two sample skeletal meshes are provided, one of which is shown in “Views” on page 8-3.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Aircraft block to your Simulink model.
2. In the Block Parameters dialog box, in the **Aircraft Parameters** tab, set the **Type** parameter to `Custom`.
3. Set the **Path to air transport mesh** parameter to either the sample mesh path or to your own air transport skeletal mesh path. The two sample skeletal mesh paths are `/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_Aircraft.SK_` and `/MathWorksAerospaceContent/Vehicles/Aircraft/Custom/Mesh/SK_HL20.SK_HL20`.
4. On the **Initial Values** tab, set the **Initial translation (in meters)** and **Initial rotation (in radians)** parameters to an array size that matches the Custom aircraft, for example, `zeros(57,3)`.

Data for Aircraft Placement for Custom Mesh

The `Custom` sample mesh origin is near its center of mass, 1.163 meters above the bottoms of the tires. To correctly place the aircraft, consider using these values.

Custom Airport Scene Placement

To place the `Air Transport` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, -1.163] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0.01984, 0]`

Custom Aircraft Altitude Sensor

For the altitude sensor in the Simulation 3D Aircraft block, use these values.

Parameter	Value
Body Z offset (m)	`1.163`
Front tire radius (m)	`0.203`
Left tire radius (m)	`0.203`
Right tire radius (m)	`0.203`

Data for Aircraft Placement for HL-20 Mesh

The `HL-20` sample mesh origin is near its center of mass, 1.385 meters above the bottoms of the tires. To correctly place the aircraft, consider using these values.

HL-20 Airport Scene Placement

To place the `HL-20` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, -1.385] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0.00901, 0]`

HL-20 Altitude Sensor

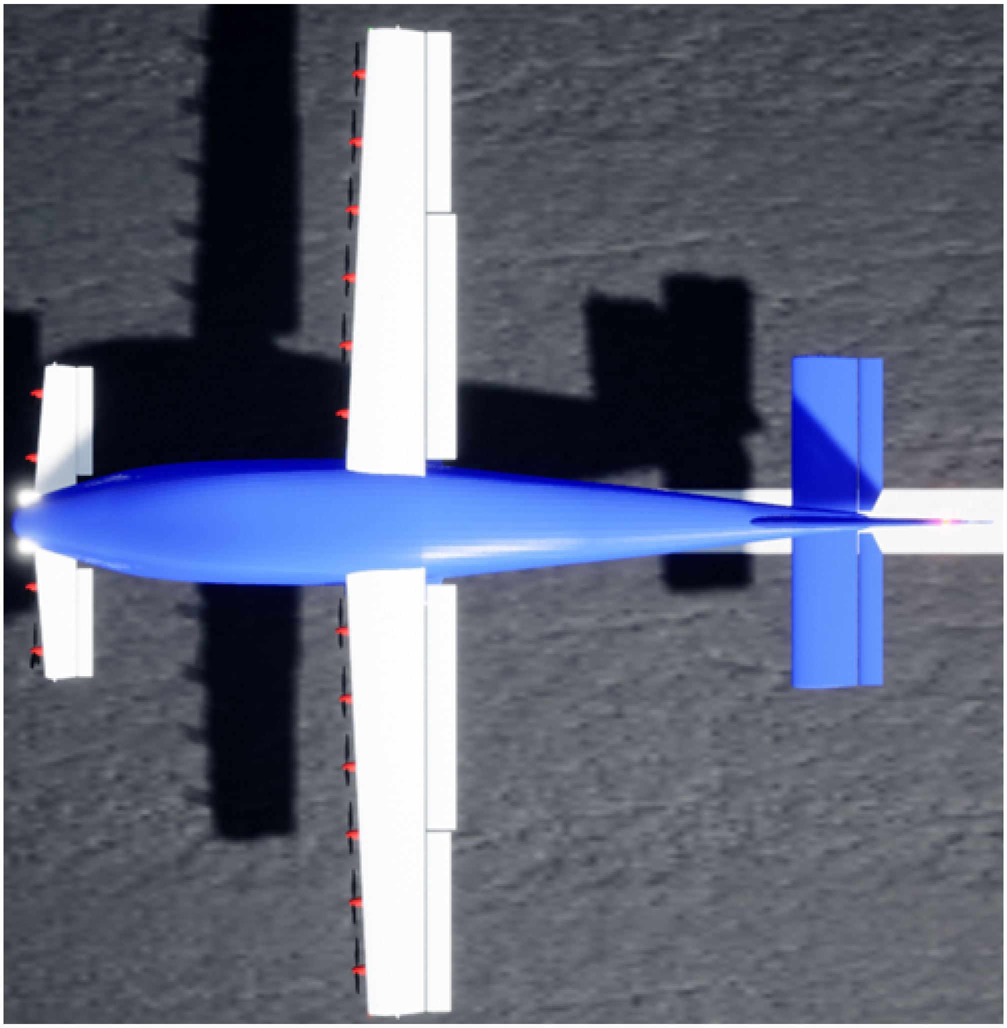
For the altitude sensor in the Simulation 3D Aircraft block, use these values.

Parameter	Value
Body Z offset (m)	`1.385`
Front tire radius (m)	`0.1745`
Left tire radius (m)	`0.2208`
Right tire radius (m)	`0.2208`

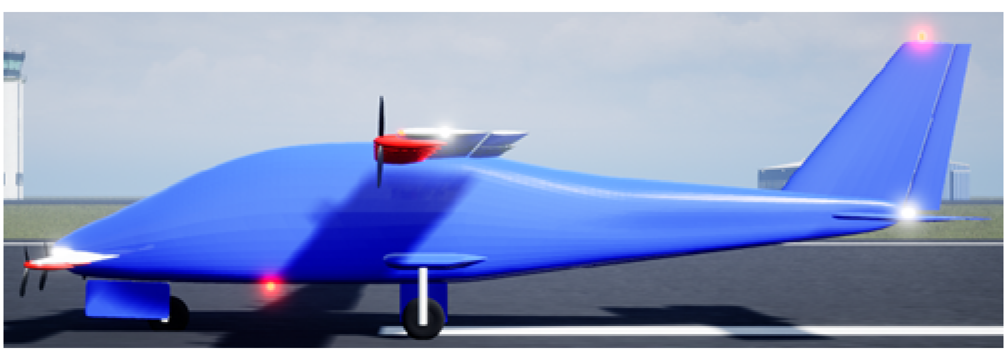
Views

This topic shows the views of the Custom aircraft. The Custom aircraft uses the same skeleton as the HL-20.

`Top-down view` — Custom aircraft top-down view
diagram

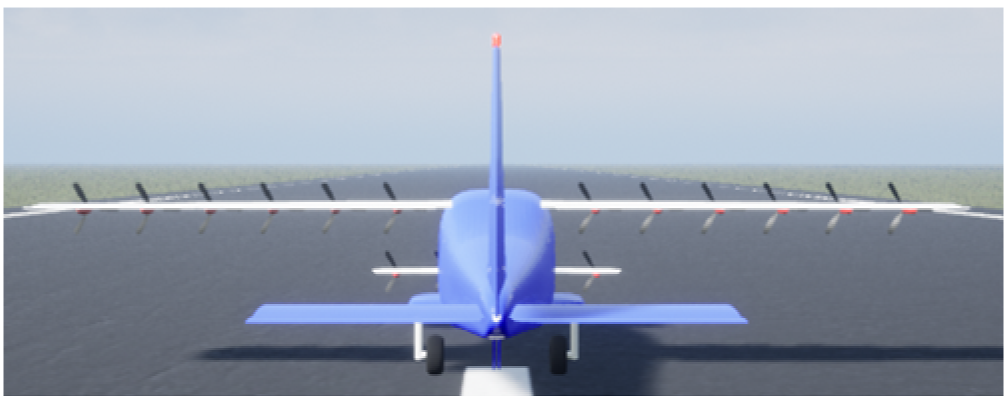


`Side view` — Custom aircraft side view
diagram



`Front view` — Custom aircraft front view
diagram

`Back view` — Custom aircraft back view
diagram



Lights and Skeleton

Lights

Light	Bone	Location in `SK_Aircraft` Mesh
Left landing	`LandingLight_L`	Left of nose
Right landing	`LandingLight_R`	Right of nose
Taxi	`NoseGear_Light`	Nose gear strut
Red navigation	`Wing1_RedNavLight`	Left wingtip of `WING1`
Green navigation	`Wing1_GreenNavLight`	Right wingtip of `WING1`
Left wingtip strobe	`Wing1_StrobeLight_L`	Left wingtip of `WING1`
Right wingtip strobe	`Wing1_StrobeLight_R`	Right wingtip of `WING1`
Tail strobe	`StrobeLight`	Top of vertical stabilizer, pointing aft
Position #1	`PosititionLight1`	Left wingtip of `WING2`
Position #2	`PosititionLight2`	Right wingtip of `WING2`
Beacon #1	`BeaconLight1`	Top of vertical stabilizer

Light	Bone	Location in SK_Aircraft Mesh
Beacon #2	BeaconLight2	Bottom of fuselage

Skeleton

- FixedWing
  - Engine1
    - Engine1_Prop
  - Engine2
    - Engine2_Prop
  - Engine3
    - Engine3_Prop
  - Engine4
    - Engine4_Prop
  - Engine5
    - Engine5_Prop
  - Engine6
    - Engine6_Prop
  - Engine7
    - Engine7_Prop
  - Engine8
    - Engine8_Prop
  - Engine9
    - Engine9_Prop
  - Engine10
    - Engine10_Prop
  - Engine11
    - Engine11_Prop
  - Engine12
    - Engine12_Prop
  - Engine13
    - Engine13_Prop
  - Engine14
    - Engine14_Prop

- Engine15
  - Engine15_Prop
- Engine16
  - Engine16_Prop
- Wing1
  - Wing1_Aileron_L
  - Wing1_Aileron_R
  - Wing1_Flap_L
  - Wing1_Flap_R
  - Wing1_Spoiler_L
  - Wing1_Spoiler_R
  - Wing1_RedNavLight
  - Wing1_GreenNavLight
  - Wing1_StrobeLight_L
  - Wing1_StrobeLight_R
- Wing2
  - Wing2_Flap_L
  - Wing2_Flap_R
- Rudder_L
- Rudder_R
- HorizStab
  - HorizStab_Elevator_L
  - HorizStab_Elevator_R
- NoseGear
  - NoseGear_Wheel
  - NoseGear_Light
- NoseGear_Door
- MainGear_L
  - MainGear_L_Wheel
- MainGear_R
  - MainGear_R_Wheel
- MainGearDoor_L
- MainGearDoor_R
- LandingLight_L
- LandingLight_R
- BeaconLight1

- `BeaconLight2`
- `StrobeLight`
- `PositionLight1`
- `PositionLight2`

Version History

Introduced in R2021b

See Also

Blocks
Simulation 3D Aircraft

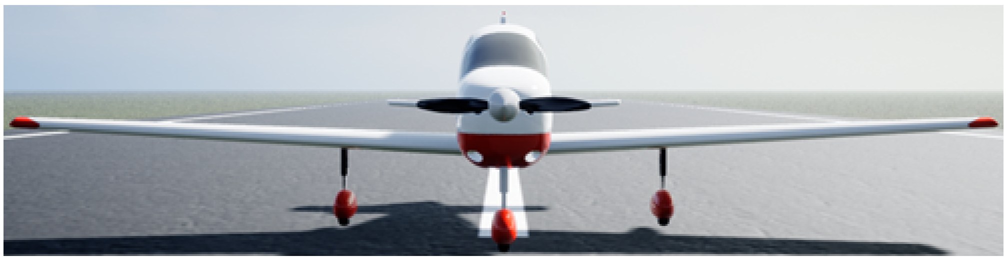
Tools
Airliner | Air Transport | General Aviation | Sky Hogg

Topics
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

General Aviation

Customizable general aviation aircraft

Description



General Aviation is one of the aircraft that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. This aircraft is designed to be customized with a user-supplied skeletal mesh; a sample skeletal mesh is provided in "Views" on page 8-16.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Aircraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Aircraft Parameters** tab, set the **Type** parameter to `General aviation`.
3. On the **Aircraft Parameters** tab, set the **Path to general aviation mesh** parameter to either the sample mesh path or to your own general aviation skeletal mesh path. The sample skeletal mesh path is `/MathWorksAerospaceContent/Vehicles/Aircraft/GeneralAviation/Mesh/SK_GeneralAviation.SK_GeneralAviation`".
4. Set the **Initial translation (in meters)** and **Initial rotation (in radians)** parameters to an array size that matches the general aviation aircraft, for example, `zeros(15,3)`.

Data for Aircraft Placement

The `General Aviation` sample mesh origin is near its center of mass, 1.646 meters above the bottoms of the tires. To correctly place the aircraft, consider using these values.

Airport Scene Placement

To place the `General Aviation` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, -1.646] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0, 0]`

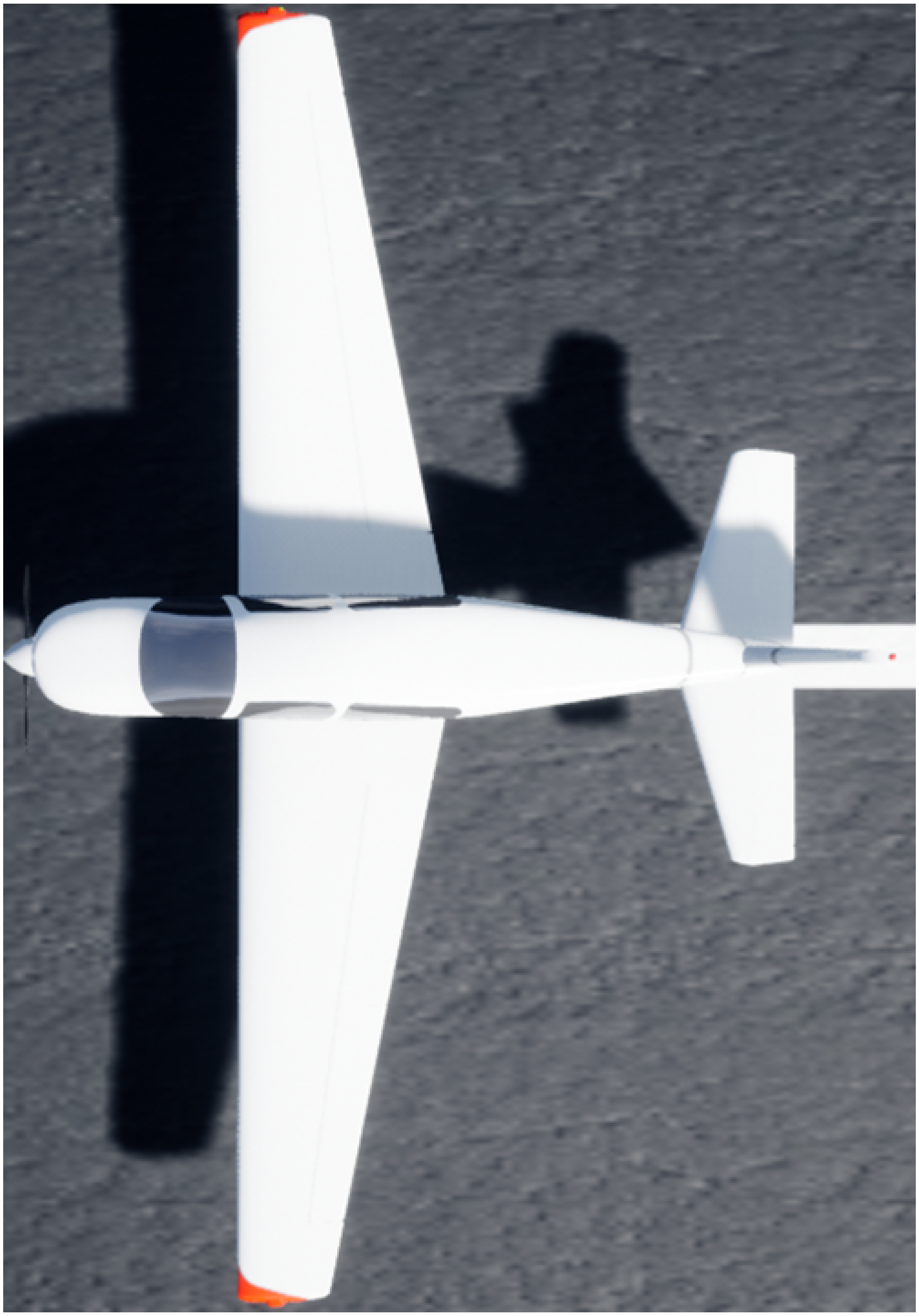
Altitude Sensor

For the altitude sensor in the Simulation 3D Aircraft block, use these values.

Parameter	Value
Body Z offset (m)	`1.6459`
Front tire radius (m)	`0.215`
Left tire radius (m)	`0.215`
Right tire radius (m)	`0.215`

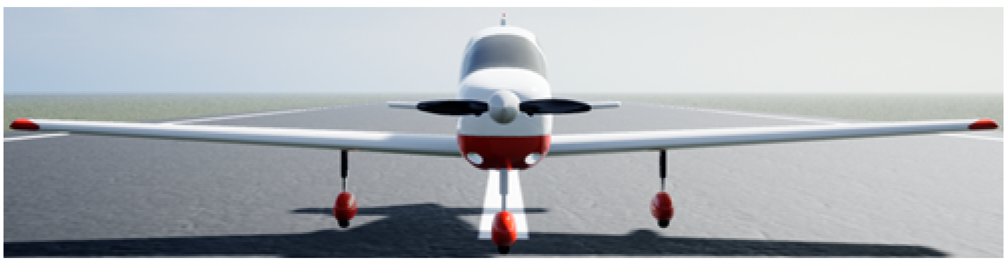
Views

`Top-down view` — General Aviation top-down view
diagram

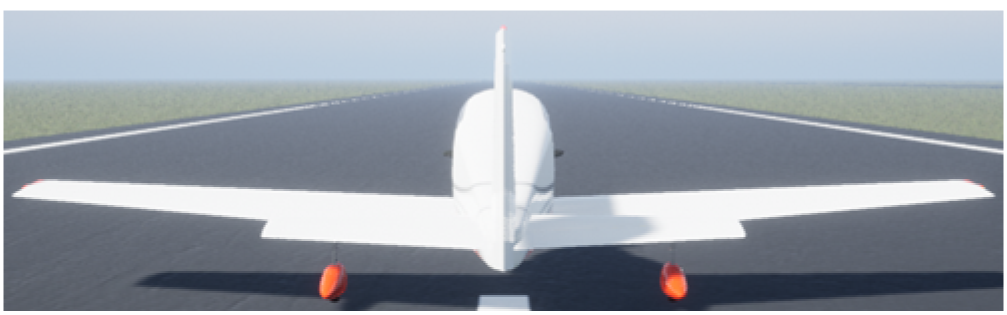


`Side view` — General Aviation side view diagram

Front view — General Aviation front view diagram



Back view — General Aviation back view diagram



Lights and Skeleton

Lights

Light	Bone	Location in SK_GeneralAviation Mesh
Left landing	LandingLight_L	Nose cowling
Right landing	LandingLight_R	Nose cowling
Taxi	NoseGear_Light	N/A
Red navigation	RedNavLight	Left wingtip

Light	Bone	Location in `SK_GeneralAviation` Mesh
Green navigation	`GreenNavLight`	Right wingtip
Left wingtip strobe	`StrobeLight_L`	Left wingtip
Right wingtip strobe	`StrobeLight_R`	Right wingtip
Tail strobe	`StrobeLight`	Top of vertical stabilizer, pointing aft
Position	`PositionLight1`	N/A
Beacon	`BeaconLight1`	Top of vertical stabilizer

Skeleton

- `GA`
  - `Engine1`
  - `Aileron_L`
  - `Aileron_R`
  - `Flap_L`
  - `Flap_R`
  - `Spoiler_L`
  - `Spoiler_R`
  - `Elevator_L`
  - `Elevator_R`
  - `Rudder`
  - `NoseGear`
    - `NoseGear_Wheel`
    - `NoseGear_Light`
  - `MainGear_L`
    - `MainGear_L_Wheel`
  - `MainGear_R`
    - `MainGear_R_Wheel`
  - `RedNavLight`
  - `GreenNavLight`
  - `StrobeLight_L`
  - `StrobeLight_R`
  - `LandingLight_L`
  - `LandingLight_R`
  - `BeaconLight1`
  - `StrobeLight`
  - `PositionLight1`

Version History

Introduced in R2021b

See Also

Blocks
Simulation 3D Aircraft

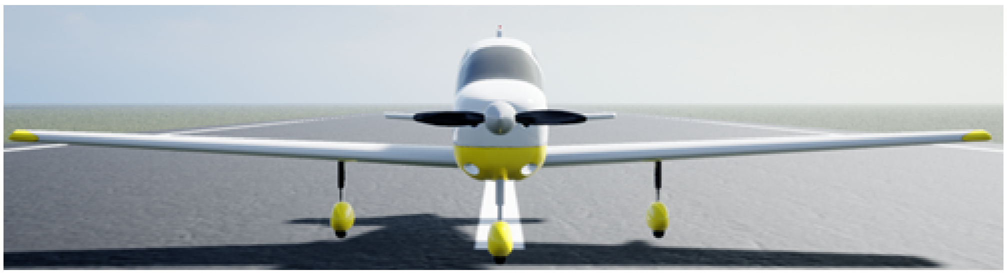
Tools
Airliner | Air Transport | Airliner | Custom | Sky Hogg

Topics
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Sky Hogg

Sky Hogg aircraft

Description



Sky Hogg is one of the aircraft that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. For detailed views of the Sky Hogg, see “Views” on page 8-22.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Aircraft block to your Simulink model.
2. In the Block Parameters dialog box, in the **Aircraft Parameters** tab, set the **Type** parameter to `Sky Hogg`.
3. Set the **Initial translation (in meters)** and **Initial rotation (in radians)** parameters to an array size that matches the Sky Hogg aircraft, for example, `zeros(11,3)`.

Data for Aircraft Placement

The `Sky Hogg` sample mesh origin is near its center of mass, 1.825 meters above the bottoms of the tires. To place the aircraft, consider using these values.

Airport Scene Placement

To place the `Sky Hogg` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, -1.825] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, 0, 0]`

Altitude Sensor

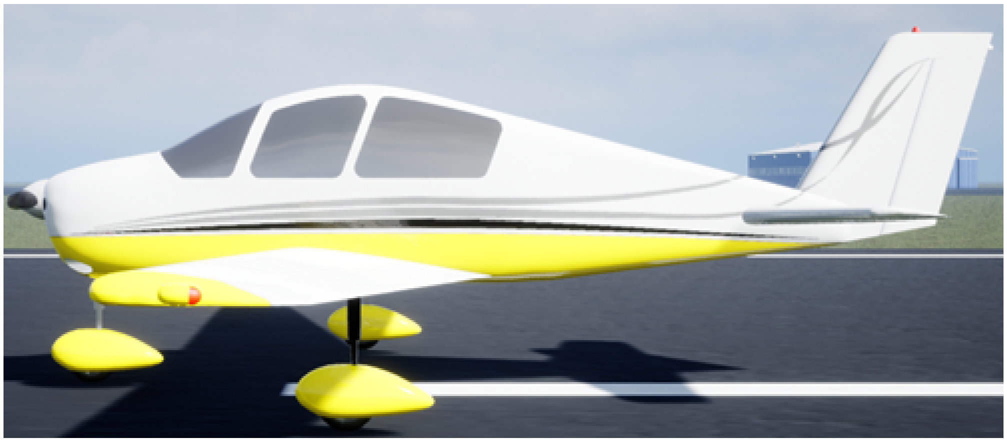
For the altitude sensor in the Simulation 3D Aircraft block, use these values.

Parameter	Value
Body Z offset (m)	`1.8249`
Front tire radius (m)	`0.215`
Left tire radius (m)	`0.215`
Right tire radius (m)	`0.215`

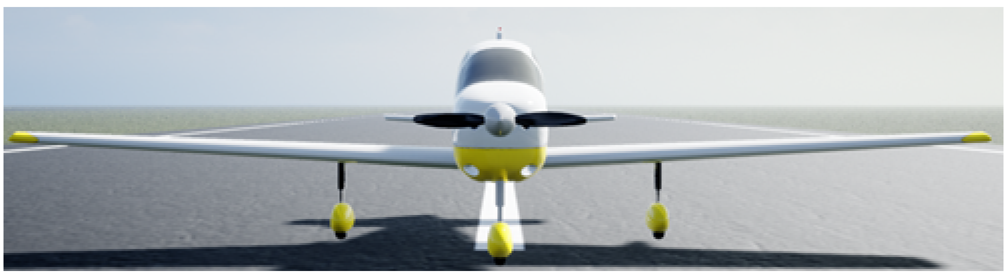
Views

`Top-down view` — Sky Hogg top-down view
diagram

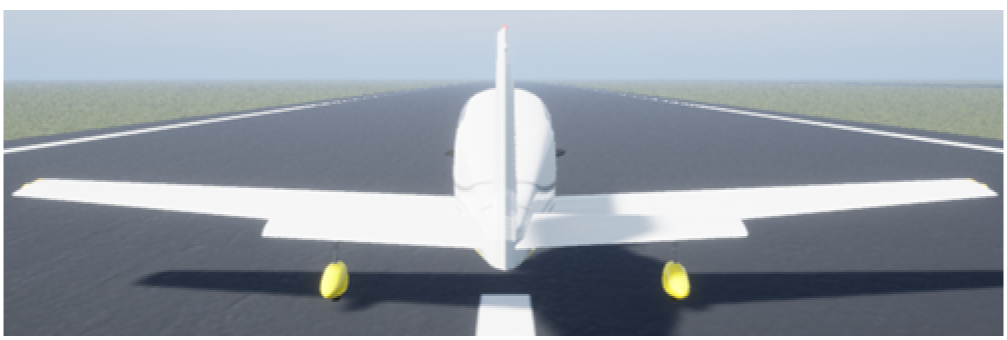
`Side view` — Sky Hogg side view
diagram



`Front view` — Sky Hogg front view
diagram



`Back view` — Sky Hogg back view
diagram



Lights and Skeleton

Lights

Light	Bone	Location in SK_SkyHogg Mesh
Landing	`SkyHogg_Landing_Light`	Nose cowling
Red navigation	`SkyHogg_RedNavigationLight`	Left wingtip

Light	Bone	Location in SK_SkyHogg Mesh
Green navigation	`SkyHogg_GreenNavigationLight`	Right wingtip
Tail strobe	`SkyHogg_NavigationLight`	Top of vertical stabilizer, pointing aft
Beacon	`SkyHogg_AntiCollisionBeacon`	Top of vertical stabilizer

Skeleton

- `SkyHogg`
  - `SkyHogg_LandingLight_Glass`
  - `SkyHogg_Wheel_R`
  - `SkyHogg_Component_Vertical`
  - `SkyHogg_Component_L`
  - `SkyHogg_Glass`
  - `SkyHogg_Body`
  - `SkyHogg_Wing_R`
  - `SkyHogg_Component_R`
  - `SkyHogg_NavigationLight`
  - `SkyHogg_Ailerons_R`
  - `SkyHogg_Flaps_R`
  - `SkyHogg_Flaps_L`
  - `SkyHogg_Ailerons_L`
  - `SkyHogg_Elevator_R`
  - `SkyHogg_Elevator_L`
  - `SkyHogg_Rudder`
  - `SkyHogg_PropellerSpinner`
    - `SkyHogg_Propeller`
  - `SkyHogg_Tire_R`
  - `SkyHogg_Tire_L`
  - `SkyHogg_NoseWheelStrut_R`
  - `SkyHogg_NoseWheelStrut_L`
  - `SkyHogg_NoseWheelStrut_F`
    - `SkyHogg_Wheel_Pants_F`
    - `SkyHogg_Tire_F`
    - `SkyHogg_Wheel_F`
  - `SkyHogg_LandingLight_Reflector`
  - `SkyHogg_RedNavigationLight`
  - `SkyHogg_GreenNavigationLight`

- `SkyHogg_NavigationLight_Glass`
- `SkyHogg_AntiCollisionBeacon`
- `SkyHogg_Wheel_L`
- `SkyHogg_Wheel_Pants_R`
- `SkyHogg_Wing_L`
- `SkyHogg_Landing_Light`

Version History

Introduced in R2021b

See Also

Blocks

Simulation 3D Aircraft

Tools

Airliner | Air Transport | Airliner | Custom | General Aviation

Topics

"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37
"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Air Transport

Customizable air transport aircraft

Description

Air Transport is one of the aircraft that you can use within the 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. This aircraft is designed to be customized with a user-supplied skeletal mesh. A sample skeletal mesh is provided in “Views” on page 8-3.

Simulating models in the 3D visualization environment requires Simulink 3D Animation.

To add this type of vehicle to the 3D simulation environment:

1. Add a Simulation 3D Aircraft block to your Simulink model.
2. In the Block Parameters dialog box, on the **Aircraft Parameters** tab, set the **Type** parameter to `Air transport`.
3. On the **Aircraft Parameters** tab, set the **Path to air transport mesh** parameter to either the sample mesh path or to your own air transport skeletal mesh path. The sample skeletal mesh path is `/MathWorksAerospaceContent/Vehicles/Aircraft/AirTransport/Mesh/SK_AirTransport.SK_AirTransport`.
4. Set the **Initial translation (in meters)** and **Initial rotation (in radians)** parameters to an array size that matches the Air Transport aircraft, for example, `zeros(30,3)`.

Data for Aircraft Placement

The `Air Transport` sample mesh origin is near its center of mass, 2.776 meters above the bottoms of the tires. To correctly place the aircraft, consider using these values.

Airport Scene Placement

To place the `Air Transport` mesh in the `Airport` scene resting on the pavement or other hard surface, which is at a *Z* of 1 centimeter, use the following body translation and rotation values.

Body Motion Ports and Parameters	Value
Translation port and **Initial translation** parameter	`[0, 0, -2.776] + [0, 0, -0.01]`
Rotation port and **Initial rotation** parameter	`[0, -0.00370, 0]`

Altitude Sensor

For the altitude sensor in the Simulation 3D Aircraft block, use these values.

Parameter	Value
Body Z offset (m)	`2.7759`
Front tire radius (m)	`0.345`
Left tire radius (m)	`0.562`
Right tire radius (m)	`0.562`

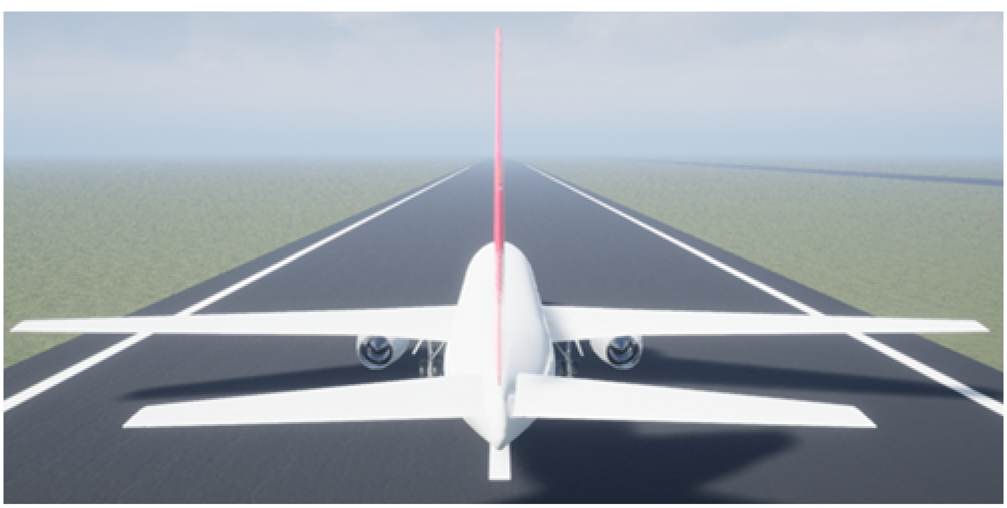
Views

`Top-down view` — Air Transport top-down view
diagram

`Side view` — Air Transport side view
diagram

`Front view` — Air Transport front view
diagram

`Back view` — Air Transport back view
diagram



Lights and Skeleton

Lights

Light	Bone	Location in `SK_AirTransport` Mesh
Left landing	`LandingLight_L`	Left wing root
Right landing	`LandingLight_R`	Right wing root
Taxi	`NoseGear_Light`	Nose gear strut
Red navigation	`RedNavLight`	Left wingtip
Green navigation	`GreenNavLight`	Right wingtip
Left wingtip strobe	`StrobeLight_L`	Left wingtip
Right wingtip strobe	`StrobeLight_R`	Right wingtip
Tail strobe	`StrobeLight`	Top of vertical stabilizer, pointing aft
Position #1	`PositionLight1`	N/A
Position #2	`PositionLight2`	N/A
Position #2	`Beacon #1`	Top of fuselage
Position #2	`Beacon #2`	Bottom of fuselage

Skeleton

- `AirTransport`
  - `Engine1`
  - `Engine2`
  - `Engine3`
  - `Engine4`
  - `Aileron_L`
  - `Aileron_R`
  - `Flap1_L`
  - `Flap1_R`
  - `Flap2_L`
  - `Flap2_R`
  - `Spoiler_L`
  - `Spoiler_R`
  - `HorizStab`
    - `HorizStab_Elevator_L`
    - `HorizStab_Elevator_R`
  - `Rudder`
  - `NoseGear`

    - NoseGear_Wheel
    - NoseGear_Light
- NoseGearDoor1
- NoseGearDoor2
- MainGear_L
    - MainGear_L_Wheel
    - MainGear_L_Link
- MainGear_L_Door1
- MainGear_L_Door2
    - MainGear_L_Door2_2
- MainGear_R
    - MainGear_R_Wheel
    - MainGear_R_Link
- MainGear_R_Door1
- MainGear_R_Door2
    - MainGear_R_Door2_2
- RedNavLight
- GreenNavLight
- StrobeLight_L
- StrobeLight_R
- LandingLight_L
- LandingLight_R
- BeaconLight1
- BeaconLight2
- StrobeLight
- PositionLight1
- PositionLight2

Version History

Introduced in R2021b

See Also

Blocks
Simulation 3D Aircraft

Tools
Airliner | Custom | Airliner | General Aviation | Sky Hogg

Topics
"Unreal Engine Simulation Environment Requirements and Limitations" on page 2-37

"Customize 3D Scenes for Aerospace Blockset Simulations" on page 4-2
"How 3D Simulation for Aerospace Blockset Works" on page 2-40

Examples

1903 Wright Flyer and Pilot with Scopes for Data Visualization

This model shows how to model the Wright Brothers' 1903 Flyer modeled in Simulink®, and Aerospace Blockset™ software. This model simulates the longitudinal motion of the Flyer in response to the pitch commands of a simulated pilot.

December 17, 2003, marked the centennial of the first powered, heavier-than-air controlled flight. This first flight happened at Kitty Hawk, North Carolina, on December 17, 1903, at 10:30 am. With a flight lasting only 12 seconds and traveling a distance of 120 feet, Orville Wright piloted his way into flight history. Three other flights occurred that day with Wilbur and Orville taking turns at the controls. Each of the flights was of increasing distance. The fourth and final flight of the day completed by Wilbur was an impressive 59 seconds traveling 852 feet. The 1903 Flyer would not take to the skies again. After the last flight of the day, the Flyer was damaged beyond repair when it was caught by a gust of wind and rolled over.

Additional information about the 1903 Flyer can be found at NASA Web Site: Re-Living The Wright Way and on MathWorks® web site: The Wright Stuff Celebrating The 1903 Flyer

A technical reference is Hooven, Frederick J., "Longitudinal Dynamics of the Wright Brothers' Early Flyers 'A Study in Computer Simulation of Flight', from The Wright Flyer An Engineering Perspective edited by Howard S. Wolko, 1987.

Note that the following warning messages are from a Simulink assertion block, used to determine if the Flyer has landed or stalled.

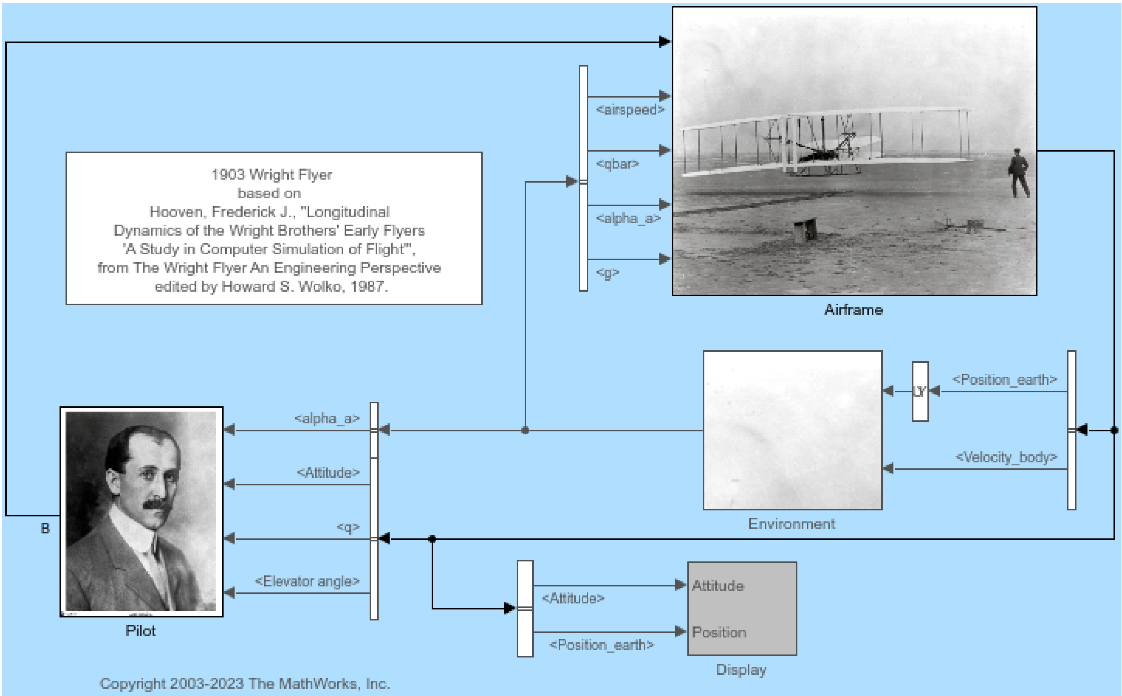
Landing

```
Warning: Assertion detected in 'aeroblk_wf_3dof_noVR/Airframe/Touch Down?/
Check Touch Down/Land?' at time 2.529176.
```

Hitting Ground

```
Warning: Assertion detected in 'aeroblk_wf_3dof_noVR/Airframe/Touch Down?/
Altitude?' at time 2.529176.
```

```
open_system('aeroblk_wf_3dof_noVR');
snapshotModel('aeroblk_wf_3dof_noVR');
```



See Also

3DOF (Body Axes) | Discrete Wind Gust Model | Incidence & Airspeed | COESA Atmosphere Model | Dynamic Pressure | WGS84 Gravity Model | Variable Transport Delay

Related Examples

- "1903 Wright Flyer" on page 3-7

Generate Unreal Engine Landscape

This example shows how to use the Unreal® Editor to create a landscape map by importing a heightmap file. Supported file formats include:

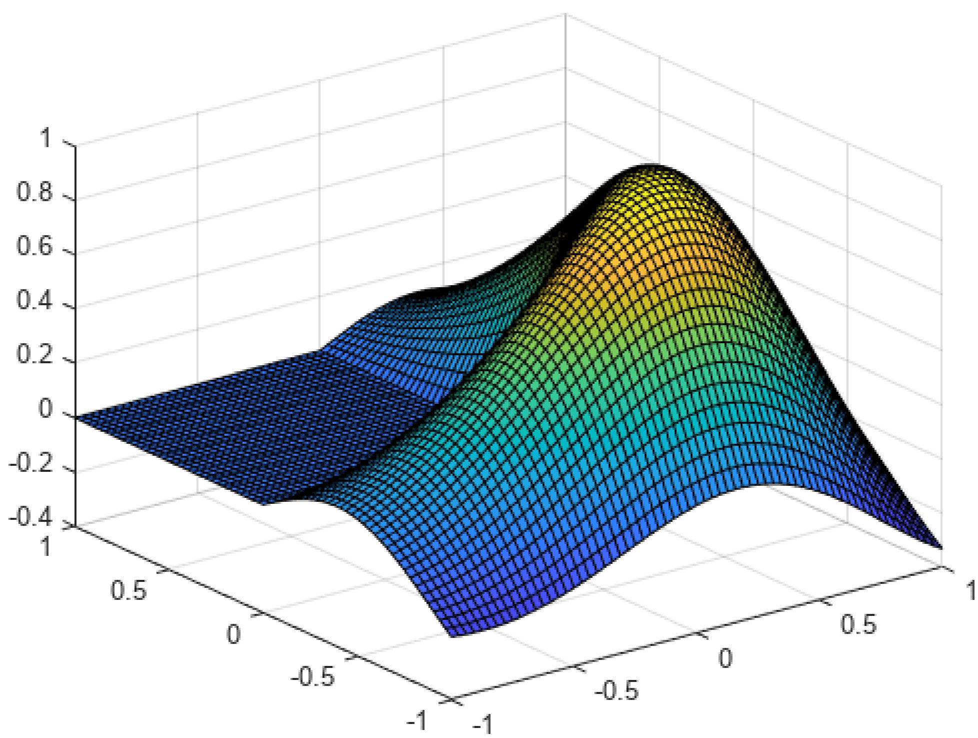
- 16-bit, grayscale PNG file
- 16-bit, grayscale RAW file in little-endian byte order

This example uses the unsigned 16-bit integer (`uint16`) format.

Create Data

Use 65-by-65 mesh points to create a new landscape using the MATLAB® shape `membrane`. The size of the mesh determines the size of the final landscape in the Unreal Editor. By default, the Unreal Editor uses a mesh of one meter by one meter, so a 65-by-65 mesh translates to a landscape size of 64 meters by 64 meters at default scale.

```
npoints = 65;
m = (npoints - 1)/2;
L = membrane(1,m);
x = -1:1/m:1;
surf(x,x',L)
```

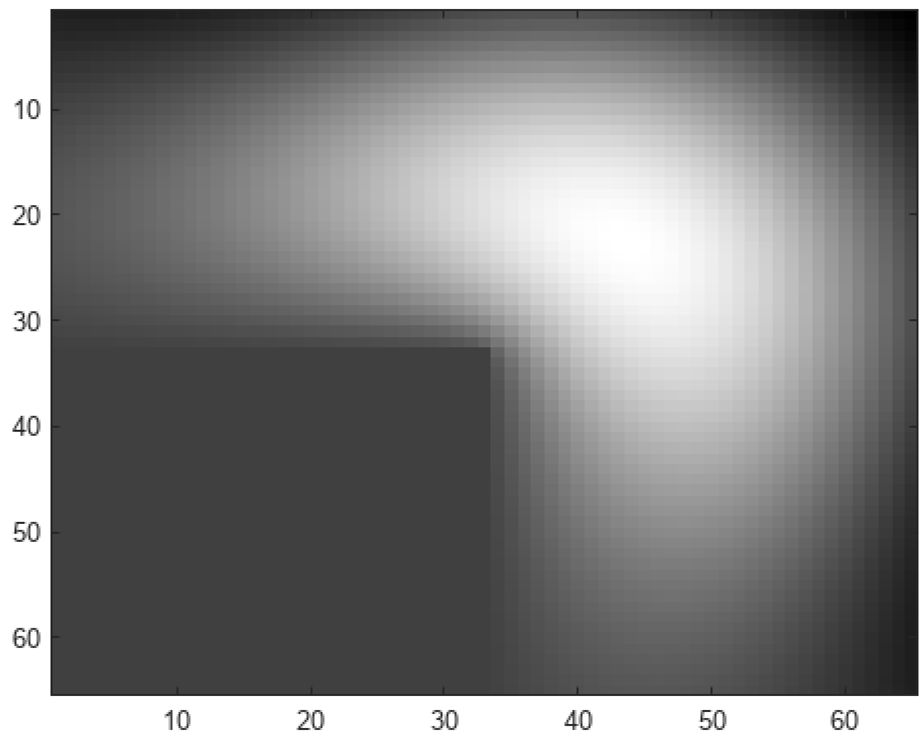


If you use sampled data instead of an analytical function, sample the heightmap data on a mesh of one by one meters. The maximum span is 8128 meters by 8128 meters. To use sampled data, choose `npoints` equal to the number of sampled points along X or Y.

Convert to Grayscale

For maximum resolution, set the range of heightmap values from the smallest (0) to the largest (65535) integer possible. Create a `uint16` grayscale heightmap using this data, and display the heightmap as an image.

```
maxL = max(L,[],"all");
minL = min(L,[],"all");
range = maxL - minL;
scale = range / 65535;
Lgray = uint16((L - minL)/scale);
imagesc(Lgray); colormap(gray)
```



Save File

Save the heightmap in a 16-bit PNG format file that you can import to the Unreal Editor.

Note: If you impose a 0:255 grayscale colormap in the export setting, you risk losing resolution.

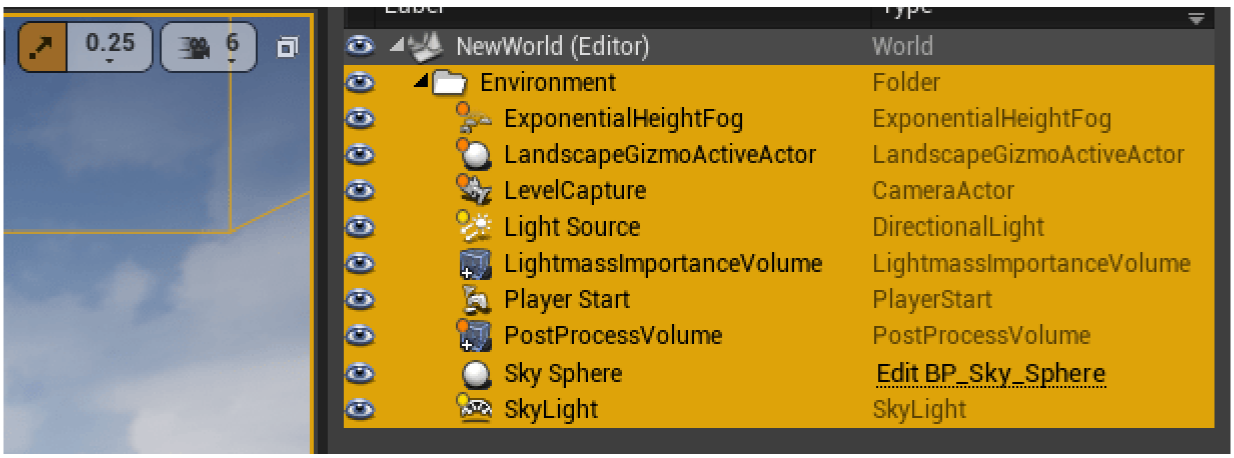
```
Lfile = 'Lgray_uint16.png';
imwrite(Lgray,Lfile,"png");
```

Create Landscape in Unreal Editor

To create the landscape in Unreal Editor, create the landscape environment, import the heightmap file, apply materials to the heightmap, and reparent the level blueprint for use in MATLAB.

Create Landscape Environment

1. Open the Unreal Editor. In the Content Browser, navigate to the desired location for the new landscape.
2. Right-click on the desired folder and select **Create Basic Asset** > **Level**.
3. Name and save the level, then double-click to open it. In the **World Outliner**, the level appears all black and contains no items.
4. Open an existing level, such as **Airport**. In the **World Outliner,** shift-select all nine items in the **Environment** folder, then right-click to **Edit** > **Copy**.
5. Open the new level again and paste the copied items into the **World Outliner**. The level now includes lighting and an atmosphere.

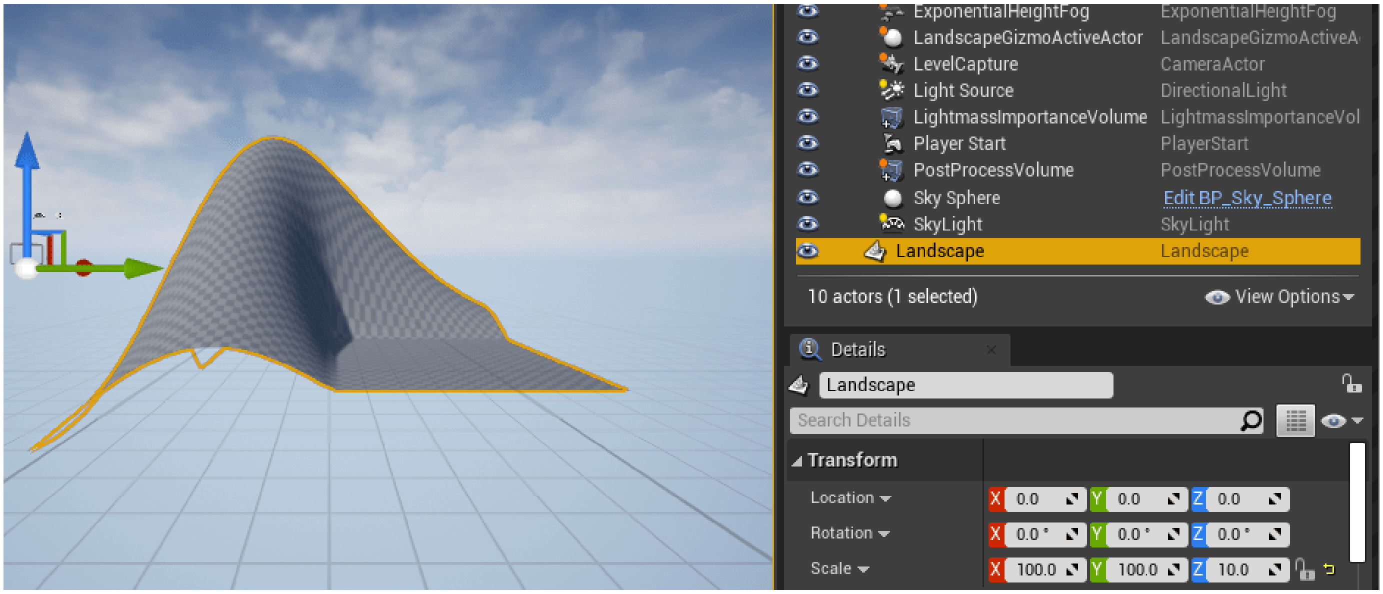


Import Heightmap File

1. On top of the viewport, select **Modes** > **Landscape** to add the **Landscape** mode toolbar to the left and bring up the **New Landscape** dialog box.
2. Click **Import from file** and select the file you created for **Heightmap File**. This action sets the **Heightmap Resolution** to the size of the loaded image, which is 65 by 65.
3. The **Scale** defaults to 100 times in all three directions. The Unreal Editor converts meters into centimeters.
4. Set **Overall Resolution** to the same value as the **Heightmap Resolution** for 1:1 scaling, with consideration for **Section Size**. The options for **Section Size** are `7x7`, `15x15`, `31x31`, `63x63`, `127x127`, and `255x255` quads. The **Overall Resolution** is a product of the **Section Size**, **Sections Per Component**, and **Number of Components** in each direction. In this example, these values are 63-by-63 sections, one section per component and one component in each direction. The overall resolution is 64-by-64.
5. Click the **Import**. A **Landscape** element is added to the **World Outliner**. Close the **Landscape** mode toolbar.
6. Change the location and scale of the landscape as desired, then press **F** to fit the selection to the window. If you set **Location** and **Scale** to `[0,0,0]` and `[100,100,10]`, respectively, and place a

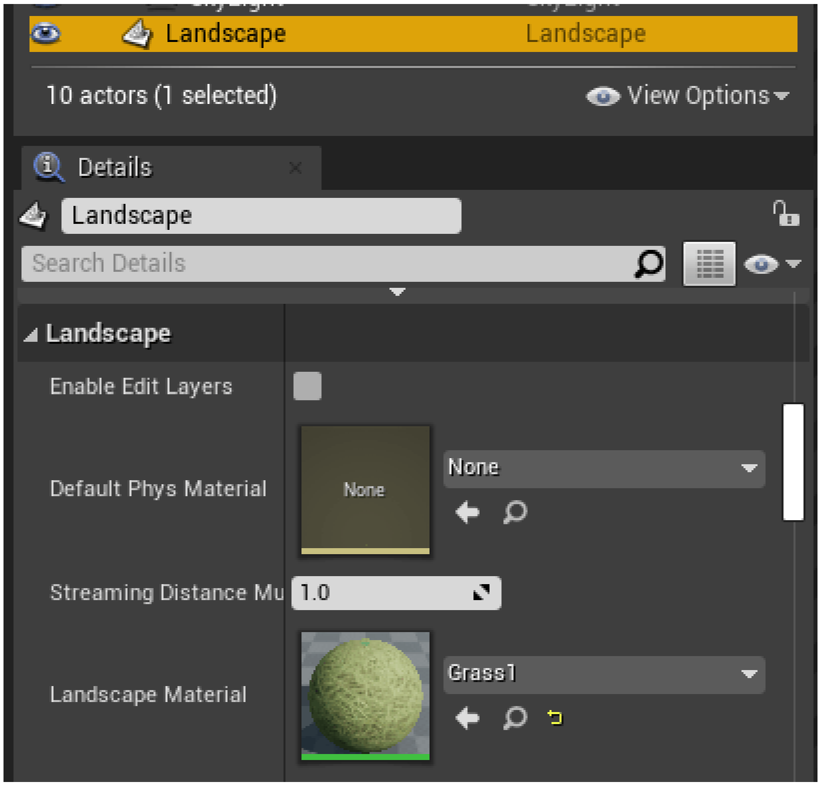
sphere in the far corner of the landscape, the coordinates of the sphere are `[6300,6300,-1840]` centimeters.

7 Adjust the **LightmassImportanceVolume** location and scale to encompass the new terrain.



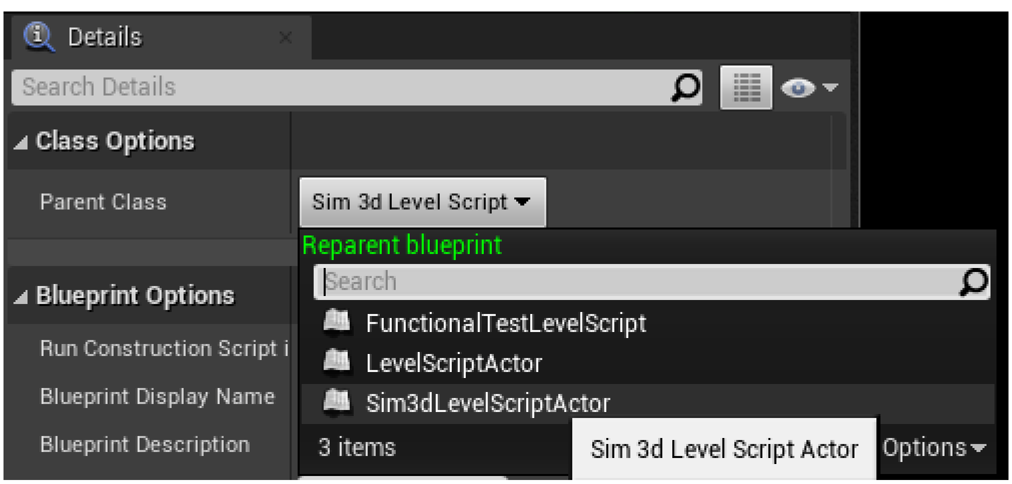
Apply Materials

In the **World Outliner**, select **Landscape**. Then, in the Details pane, find **Landscape Material**. An empty Landscape Material value results in a checkered gray image. To change the landscape appearance, select a material to use, for example, **MathWorksAerospaceContent** > **Environment** > **Apple_Hill** > **Materials** > **Grass_1**. Click the material to select it, then click the left-pointing the **Landscape Material** arrow to apply the material.

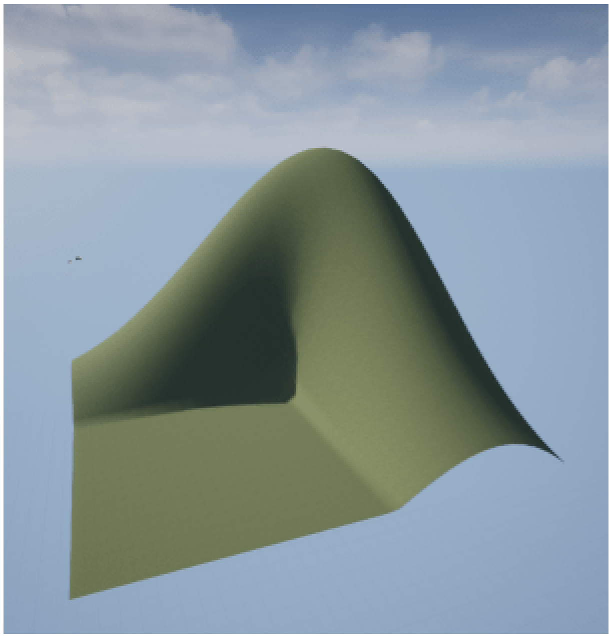


Reparent Blueprint

Click **Blueprints** at the top of the viewing window and select **Open Level Blueprint**. In the dialog box, click **Class Settings**. The first item in the **Details** pane is **Class Options > Parent Class**. Set **Parent Class** to **Sim 3d Level Script Actor**. This setting enables this level to be used in MATLAB. Save and exit the dialog.



The new landscape map is now ready to use.



See Also

External Websites

- Unreal Engine Documentation

1903 Wright Flyer and Pilot with Simulink 3D Animation

This model shows how to model the Wright Brothers' 1903 Flyer modeled in Simulink®, Aerospace Blockset™ and Simulink® 3D Animation™ software. This model simulates the longitudinal motion of the Flyer in response to the pitch commands of a simulated pilot.

December 17, 2003, marked the centennial of the first powered, heavier-than-air controlled flight. This first flight happened at Kitty Hawk, North Carolina, on December 17, 1903, at 10:30 am. With a flight lasting only 12 seconds and traveling a distance of 120 feet, Orville Wright piloted his way into flight history. Three other flights occurred that day with Wilbur and Orville taking turns at the controls. Each of the flights was of increasing distance. The fourth and final flight of the day completed by Wilbur was an impressive 59 seconds traveling 852 feet. The 1903 Flyer would not take to the skies again. After the last flight of the day, the Flyer was damaged beyond repair when it was caught by a gust of wind and rolled over.

Additional information about the 1903 Flyer can be found at NASA Web Site: Re-Living The Wright Way and on MathWorks® web site: The Wright Stuff Celebrating The 1903 Flyer

A technical reference is Hooven, Frederick J., "Longitudinal Dynamics of the Wright Brothers' Early Flyers 'A Study in Computer Simulation of Flight', from The Wright Flyer An Engineering Perspective edited by Howard S. Wolko, 1987.

Note that the following warning messages are from a Simulink assertion block, used to determine if the Flyer has landed or stalled.

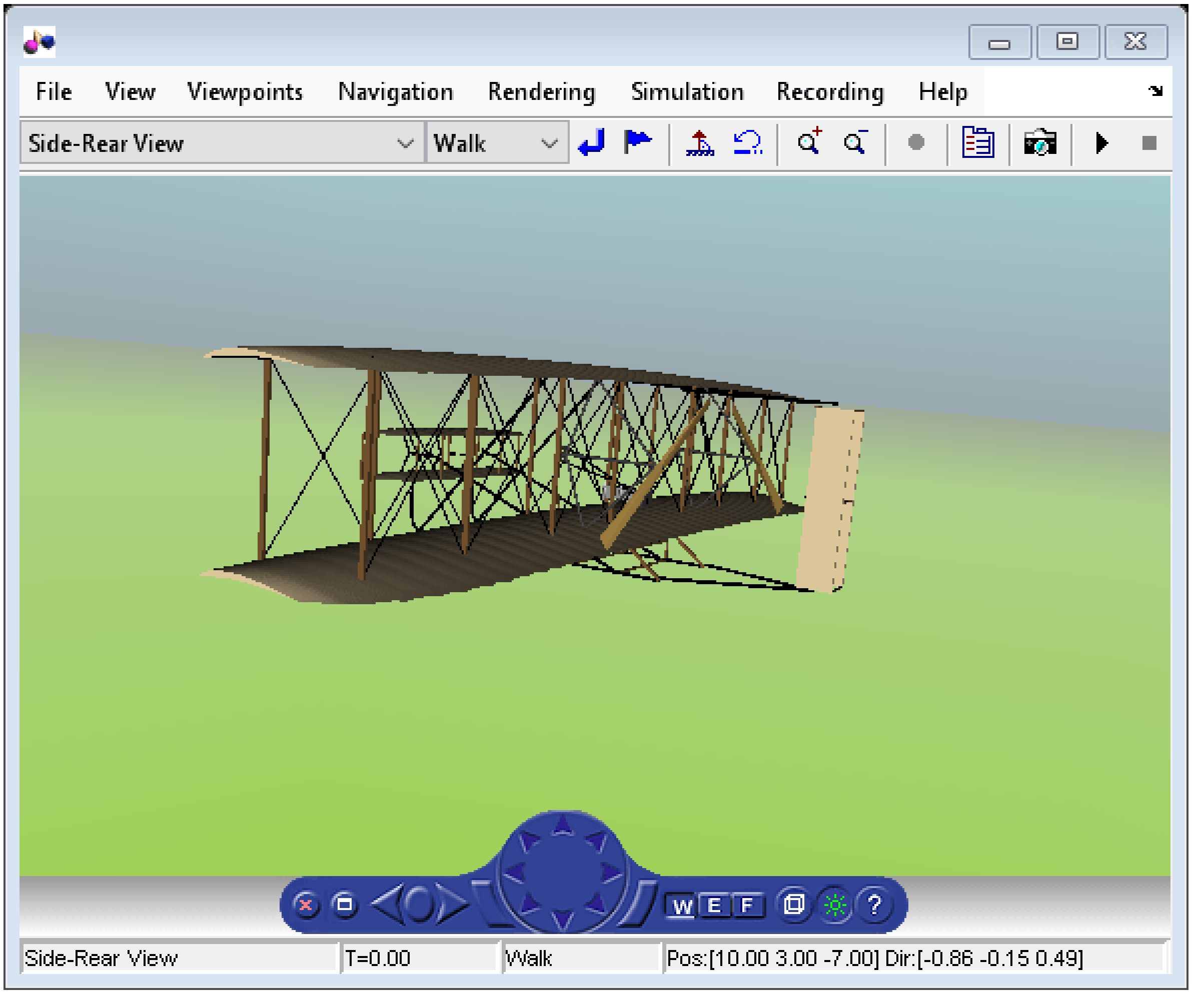
Landing

```
Warning: Assertion detected in 'aeroblk_wf_3dof/Airframe/Touch Down?/Check
Touch Down/Land?' at time 2.529176.
```

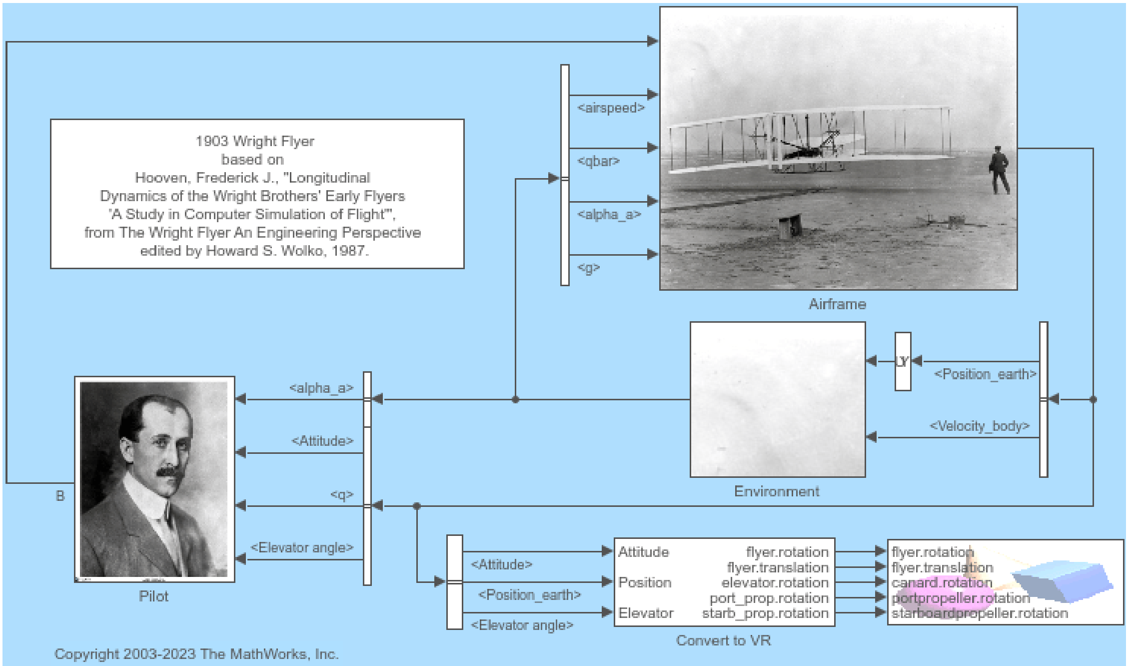
Hitting Ground

```
Warning: Assertion detected in 'aeroblk_wf_3dof/Airframe/Touch Down?/
Altitude?' at time 2.529176.
```

```
open_system('aeroblk_wf_3dof');
```




```
snapshotModel('aeroblk_wf_3dof');
```



See Also

3DOF (Body Axes) | Incidence & Airspeed | COESA Atmosphere Model | Dynamic Pressure | WGS84 Gravity Model

Related Examples

- “1903 Wright Flyer” on page 3-7

Fly the De Havilland Beaver

This model shows how to model the De Havilland Beaver using Simulink® and Aerospace Blockset™ software. It also shows how to use a pilot's joystick to fly the De Havilland Beaver. This model has been color-coded to aid in locating Aerospace Blockset blocks. The red blocks are Aerospace Blockset blocks, the orange blocks are subsystems containing additional Aerospace Blockset blocks, and the white blocks are Simulink blocks.

The De Havilland Beaver model includes the airframe dynamics and aerodynamics. Effects of the environment are also modeled, such as wind profiles for the landing phase. Visualization for this model is done via an interface to FlightGear, a open source flight simulator package.

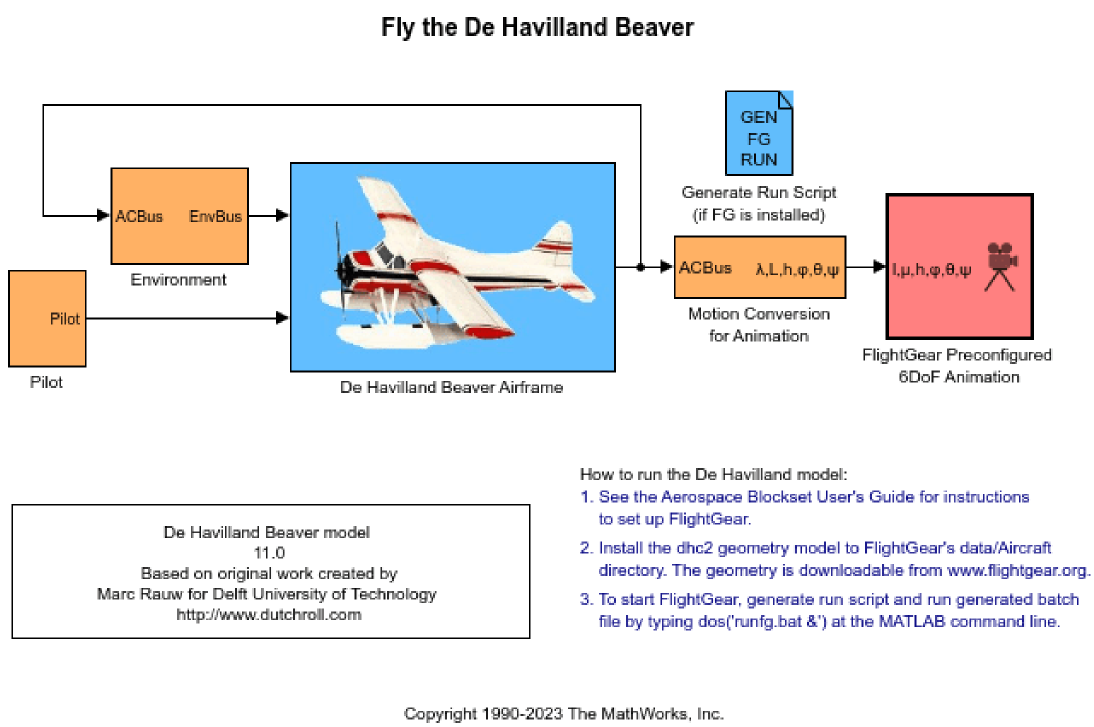
For more information on the FlightGear interface, read these documentation topics:

"Flight Simulator Interface" on page 2-20

"Work with the Flight Simulator Interface" on page 2-24

"Run the HL-20 Example with FlightGear" on page 2-32

```
open_system('asbdhc2');
snapshotModel('asbdhc2');
```



The De Havilland Beaver was first flown in 1947. Today it is still prized by pilots for its reliability and versatility. The De Havilland Beaver can be operated on wheels, skis or float landing gear.

Speed maximum: 110 kts, Altitude maximum: 10,000 ft, Range maximum: 400 nm, Load: 6 passengers, Crew: 1 member.

See Also

6DOF (Quaternion) | COESA Atmosphere Model | Discrete Wind Gust Model | Dryden Wind Turbulence Model (Continuous) | FlightGear Preconfigured 6DoF Animation | Generate Run Script | Wind Shear Model

Related Examples

- "Flight Simulator Interface" on page 2-20
- "Work with the Flight Simulator Interface" on page 2-24
- "Run the HL-20 Example with FlightGear" on page 2-32

Lightweight Airplane Design

This model shows how to use MathWorks® products to address the technical and process challenges of aircraft design using the design of a lightweight aircraft.

The design process is iterative; you will try many vehicle configurations before selecting the final one. Ideally, you perform iterations before building any hardware. The challenge is to perform the iterations quickly. Typically, different groups work on different steps of the process. Effective collaboration among these groups and the right set of tools are essential to addressing this challenge.

Defining Vehicle Geometry

The geometry of this lightweight aircraft is from reference 1. The original design objective for this geometry is a four-seat general aviation aircraft that is safe, simple to fly, and easily maintainable with specific mission and performance constraints. For more details on these constraints, see reference 1.

Potential performance requirements for this aircraft include:

- Level cruise speed
- Acceptable rate of climb
- Acceptable stall speed

For the aircraft flight control, rate of climb is the design requirement and assumed to be greater than 2 meters per second (m/s) at 2000 meters.

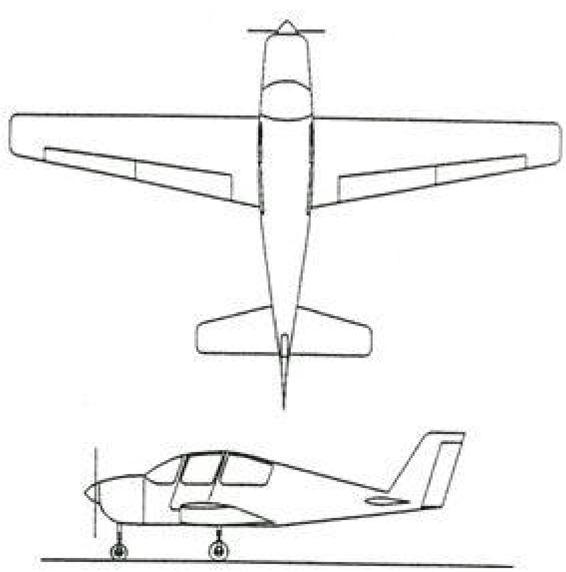


Figure 1: Lightweight four-seater monoplane [1].

Determining Vehicle Aerodynamic Characteristics

The aircraft's geometrical configuration determines its aerodynamic characteristics, and therefore its performance and handling qualities. Once you choose the geometric configuration, you can obtain the aerodynamic characteristics by means of:

- Analytical prediction
- Wind tunnel testing of the scaled model or a full-sized prototype
- Flight tests

While wind tunnel tests and flight tests provide high-fidelity results, they are expensive and time-consuming, because they must be performed on the actual hardware. It is best to use these methods when the aircraft's geometry is finalized. **Note**: Analytical prediction is a quicker and less expensive way to estimate aerodynamic characteristics in the early stages of design.

This example uses Digital Datcom, a popular software program, for analytical prediction. The U.S. Air Force developed it as a digital version of its Data Compendium (DATCOM). This software is publicly available.

To start, create a Digital Datcom input file that defines the geometric configuration of our aircraft and the flight conditions needed to obtain the aerodynamic coefficients.

```
type asbSkyHoggDatcom.in
```

```
 $FLTCON NMACH=4.0,MACH(1)=0.1,0.2,0.3,0.35$
 $FLTCON NALT=8.0,ALT(1)=1000.0,3000.0,5000.0,7000.0,9000.0,
   11000.0,13000.0,15000.0$
 $FLTCON NALPHA=10.,ALSCHD(1)=-16.0,-12.0,-8.0,-4.0,-2.0,0.0,2.0,
   ALSCHD(8)=4.0,8.0,12.0,LOOP=2.0$
 $OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
 $SYNTHS XCG=7.9,ZCG=-1.4,XW=6.1,ZW=0.0,ALIW=1.1,XH=20.2,
    ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
 $BODY NX=10.0,
    X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
    R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
 $WGPLNF CHRDTP=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
    TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
 $HTPLNF CHRDTP=2.3,SSPNE=5.7,SSPN=6.625,CHRDR=0.25,SAVSI=11.0,
    CHSTAT=1.0,TWISTA=0.0,TYPE=1.0$
 $VTPLNF CHRDTP=2.7,SSPNE=5.0,SSPN=5.2,CHRDR=5.3,SAVSI=31.3,
    CHSTAT=0.25,TWISTA=0.0,TYPE=1.0$
 $SYMFLP NDELTA=5.0,DELTA(1)=-20.,-10.,0.,10.,20.,PHETE=.0522,
    CHRDFI=1.3,
    CHRDFO=1.3,SPANFI=.1,SPANFO=6.0,FTYPE=1.0,CB=1.3,TC=.0225,
    PHETEP=.0391,NTYPE=1.$
NACA-W-4-0012
NACA-H-4-0012
NACA-V-4-0012
CASEID SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG
DAMP
NEXT CASE
```

Digital Datcom provides the vehicle's aerodynamic stability and control derivatives and coefficients at specified flight conditions. Flight control engineers can gain insight into the vehicle's performance and handling characteristics by examining stability and control derivatives. You must import this data into the MATLAB® technical computing environment for analysis. Normally, this is a manual process.

With the Aerospace Toolbox software, you can bring multiple Digital Datcom output files into the MATLAB technical computing environment with just one command. There is no need for manual input. Each Digital Datcom output is imported into the MATLAB technical computing environment as a cell array of structures, with each structure corresponding to a different Digital Datcom output file.

After importing the Digital Datcom output, run multiple configurations through Digital Datcom and compare the results in the MATLAB technical computing environment.

```
alldata = datcomimport({'asbSkyHoggDatcom.out' 'asbSkyHoggDatcom_old.out'}, true, 0);
```

In our model, you need to check whether the vehicle is inherently stable. To do this, you can use Figure 2 to check whether the pitching moment described by the corresponding coefficient, Cm, provides a restoring moment for the aircraft. A restoring moment returns the aircraft angle of attack to zero.

In configuration 1 (Figure 2), Cm is negative for some angles of attack less than zero. This means that this configuration will not provide a restoring moment for those negative angles of attack and will not provide the flight characteristics that are desirable. Configuration 2 fixes this problem by moving the center of gravity rearward. Shifting the center of gravity produces a Cm that provides a restoring moment for all negative angles of attack.

```
h1 = figure;
plot(alldata{2}.alpha,alldata{2}.cm(:,1),alldata{1}.alpha,alldata{1}.cm(:,1));
axis([-4 8 -0.3 0.1]);
legend('Configuration 1','Configuration 2');
xlabel('Angle of Attack, \alpha (degrees)');
ylabel('Pitching Moment Coefficient, C_m');
grid;
```

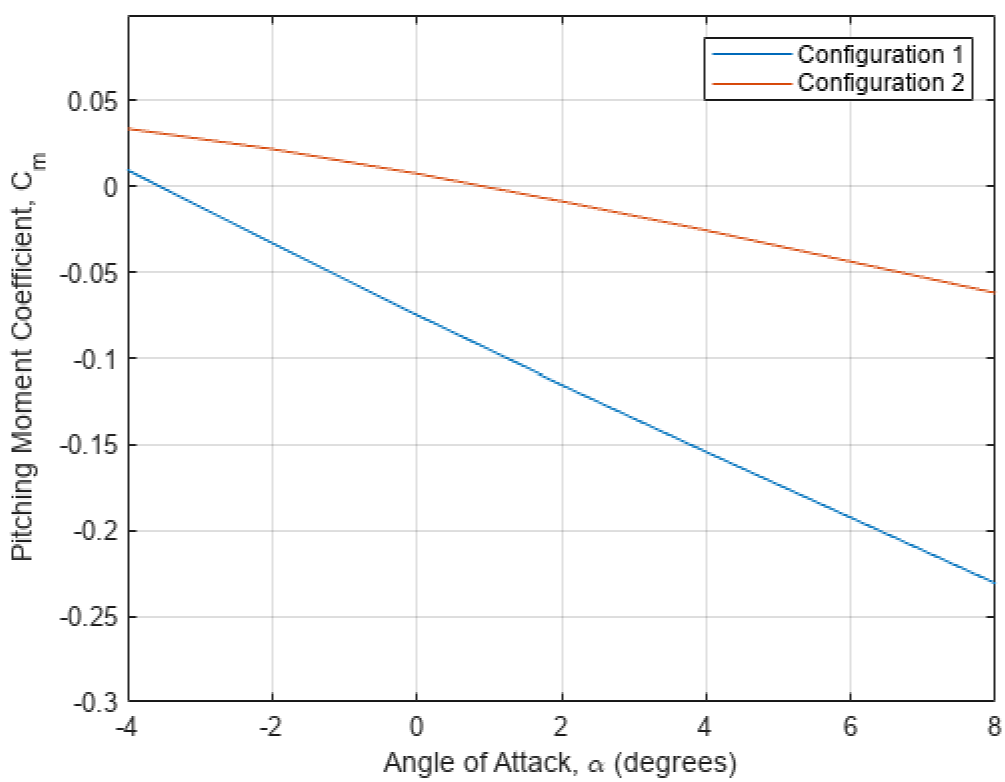


Figure 2: Visual analysis of Digital Datcom pitching moment coefficients.

```
close (h1);
```

Creating Flight Vehicle Simulation

Once you determine aerodynamic stability and control derivatives, you can build an open-loop plant model to evaluate the aircraft longitudinal dynamics. Once the model is complete, you can show it to colleagues, including those who do not have Simulink® software, by using Simulink Report Generator™ software to export the model to a Web view. A Web view is an interactive HTML replica of the model that lets you navigate model hierarchy and check the properties of subsystems, blocks, and signals.

A typical plant model includes the following components:

- **Equations of motion**: Calculate vehicle position and attitude from forces and moments.
- **Forces and moments**: Calculate aerodynamic, gravity, and thrust forces and moments.
- **Actuator positions**: Calculate displacements based on actuator commands.
- **Environment**: Include environmental effects of wind disturbances, gravity, and atmosphere.
- **Sensors**: Model the behavior of the measurement devices.

You can implement most of this functionality using Aerospace Blockset™ blocks. This model highlights subsystems containing Aerospace Blockset blocks in orange. It highlights Aerospace Blockset blocks in red.

```
open_system('asbSkyHogg');
snapshotModel('asbSkyHogg');
```

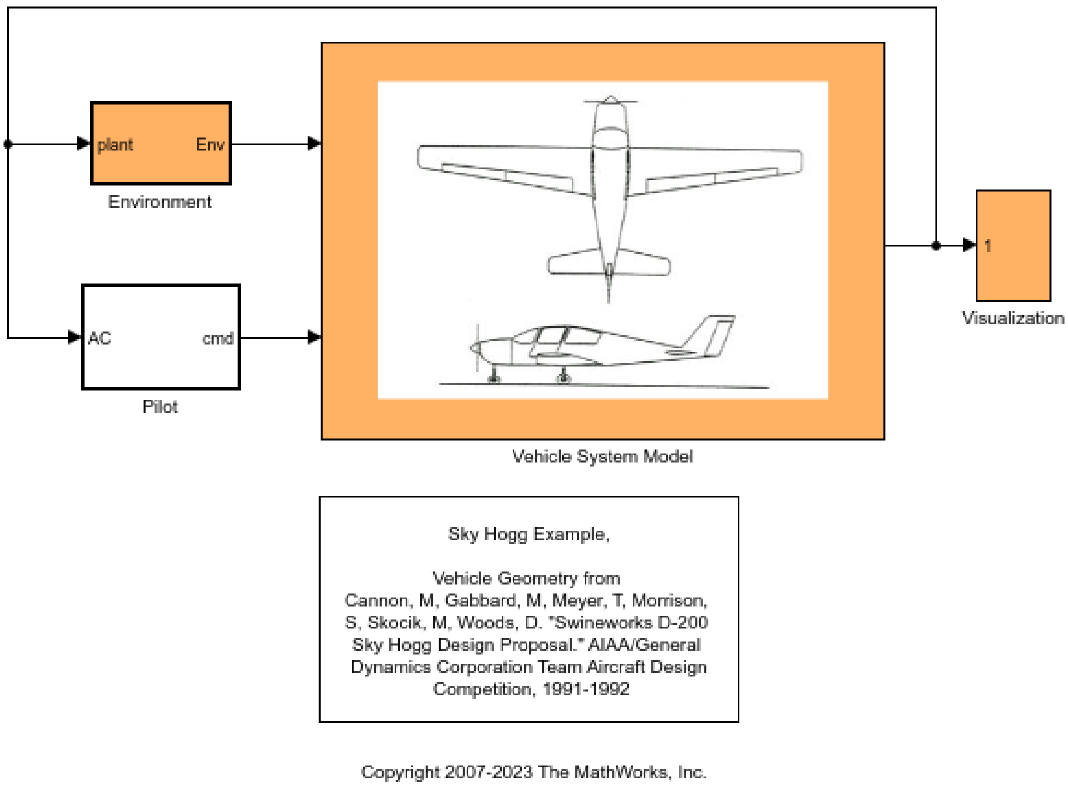


Figure 3: Top Level of Lightweight Aircraft Model

You begin by building a plant model using a 3DOF block from the Equations of Motion library in the Aerospace Blockset library (Figure 4). This model helps us determine whether the flight vehicle is longitudinally stable and controllable. Design our subsystem to have the same interface as a six degrees-of-freedom (DOF) version. When you are satisfied with three DOF performance, stability, and controllability, you can implement the six DOF version, iterating on the other control surface geometries until you achieve the desired behavior from the aircraft.

```
open_system('asbSkyHogg/Vehicle System Model/Vehicle/3DOF to 6DOF');
snapshotModel('asbSkyHogg/Vehicle System Model/Vehicle/3DOF to 6DOF');
```

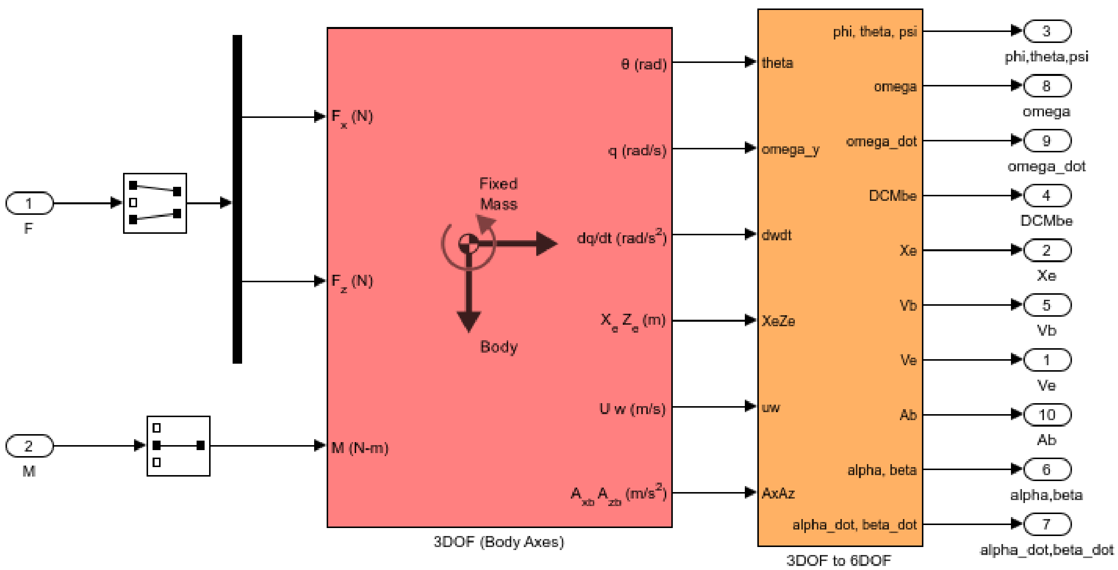


Figure 4: Equations of Motion implemented using 3DoF Euler block from the Aerospace Blockset library.

To calculate the aerodynamic forces and moments acting on our vehicle, use a Digital Datcom Forces and Moments block from the Aerospace Blockset library (Figure 5). This block uses a structure that Aerospace Toolbox creates when it imports aerodynamic coefficients from Digital Datcom.

For some Digital Datcom cases, dynamic derivative have values for only the first angle of attack. The missing data points can be filled with the values for the first angle of attack, since these derivatives are independent of angle of attack. To see example code of how to fill in missing data in Digital Datcom data points, you can examine the asbPrepDatcom function.

```
open_system('asbSkyHogg/Vehicle System Model/Vehicle/Aerodynamics');
snapshotModel('asbSkyHogg/Vehicle System Model/Vehicle/Aerodynamics');
```

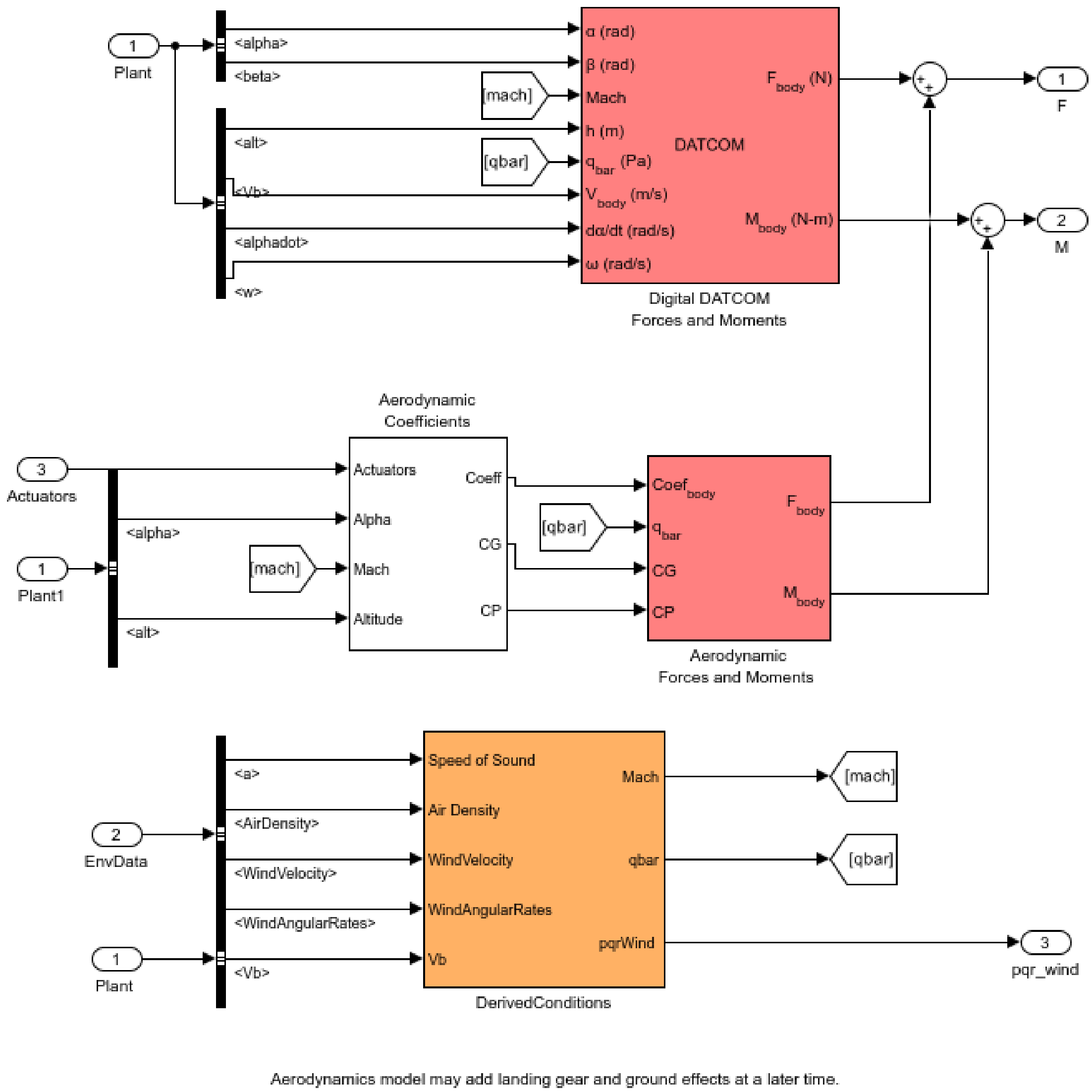


Figure 5: Aerodynamic Forces and Moments implemented in part with the Aerospace Blockset Digital Datcom Forces and Moment block.

You also use Aerospace Blockset blocks to create actuator, sensor, and environment models (Figures 6, 7, and 8, respectively). **Note**: In addition to creating the following parts of the model, use standard Aerospace Blockset blocks to ensure that you convert from body axes to wind axes and back correctly.

```
open_system('asbSkyHogg/Vehicle System Model/Vehicle/AirframeActuators');
snapshotModel('asbSkyHogg/Vehicle System Model/Vehicle/AirframeActuators');
```

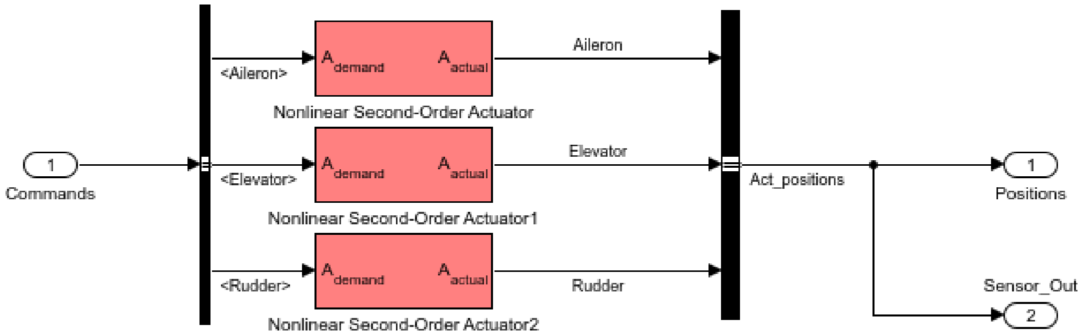


Figure 6: Implementation of actuator models using Aerospace Blockset blocks.

```
open_system('asbSkyHogg/Vehicle System Model/Flight Sensors');
snapshotModel('asbSkyHogg/Vehicle System Model/Flight Sensors');
```

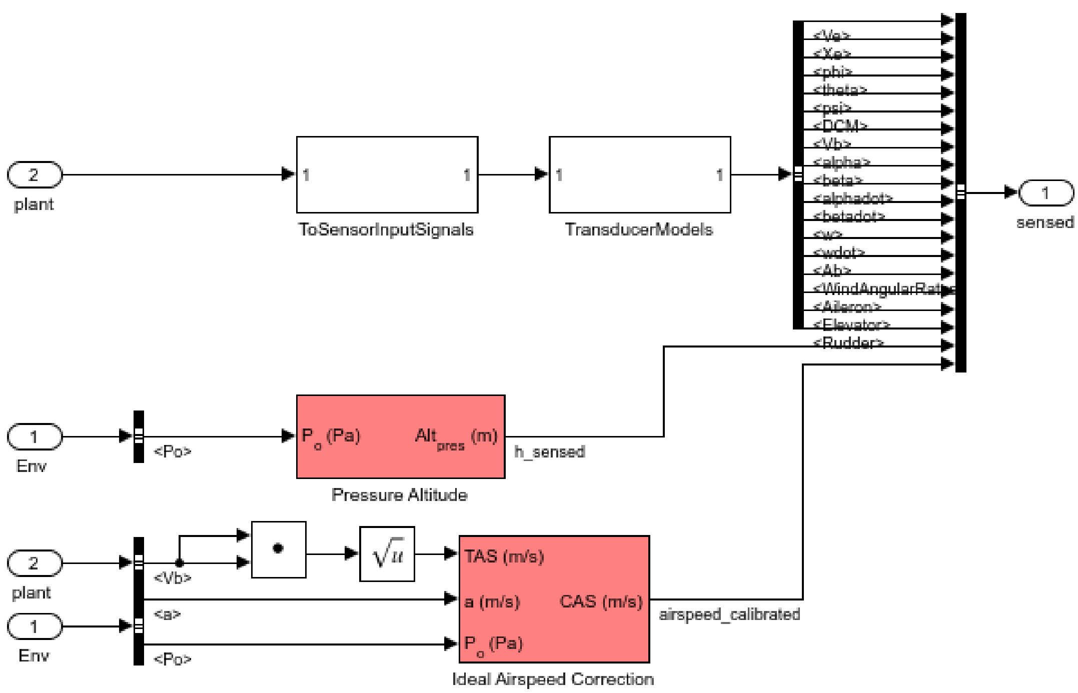


Figure 7: Implementation of flight sensor model using Aerospace Blockset blocks.

```
open_system('asbSkyHogg/Environment');
snapshotModel('asbSkyHogg/Environment');
```

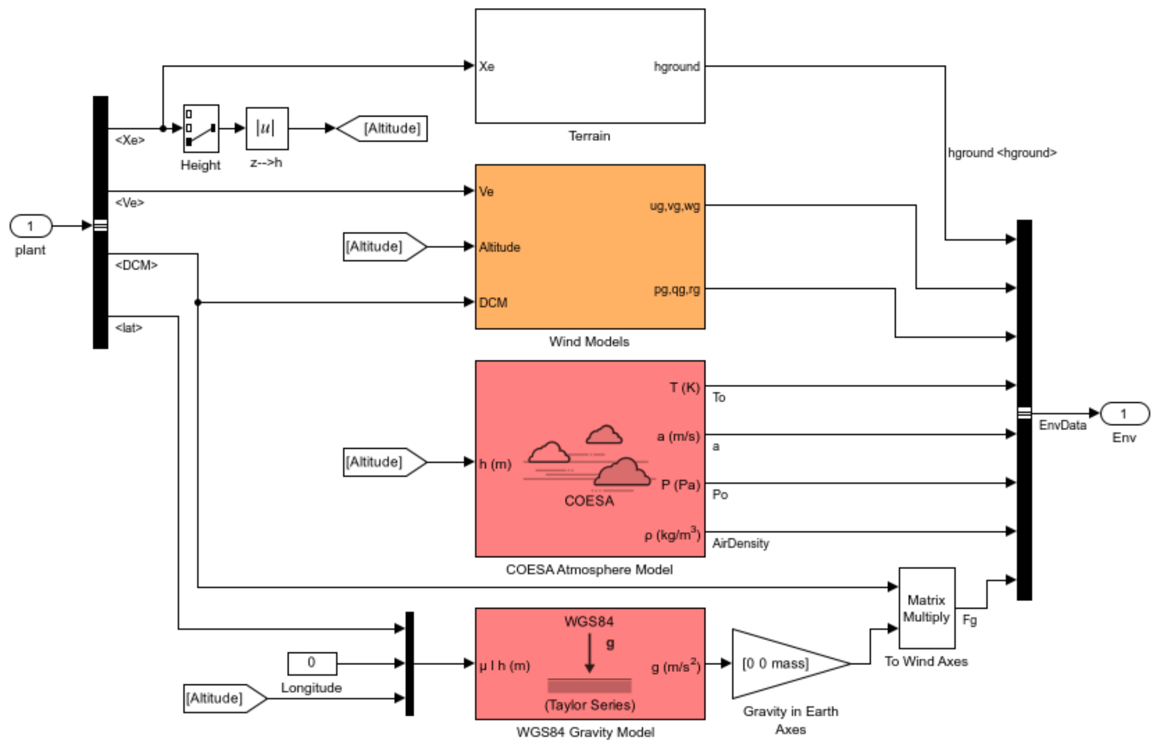


Figure 8: Environmental effects of wind, atmosphere, and gravity using Aerospace Blockset blocks.

Designing Flight Control Laws

Once you create the Simulink plant model, design a longitudinal controller that commands elevator position to control altitude. The traditional two-loop feedback control structure chosen for this design (Figure 9) has an outer loop for controlling altitude (compensator C1 in yellow) and an inner loop for controlling pitch angle (compensator C2 in blue). Figure 10 shows the corresponding controller configuration in our Simulink model.

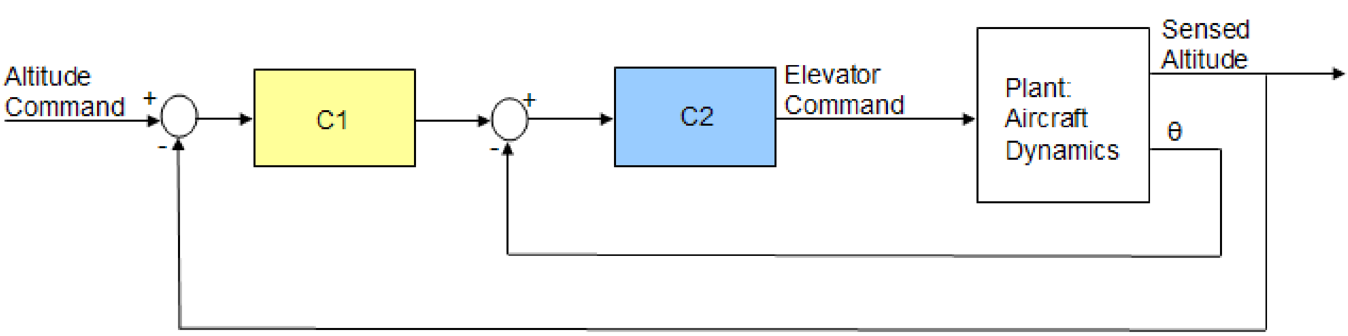


Figure 9: Structure of the longitudinal controller.

```
open_system('asbSkyHogg/Vehicle System Model/Avionics/Autopilot');
snapshotModel('asbSkyHogg/Vehicle System Model/Avionics/Autopilot');
```

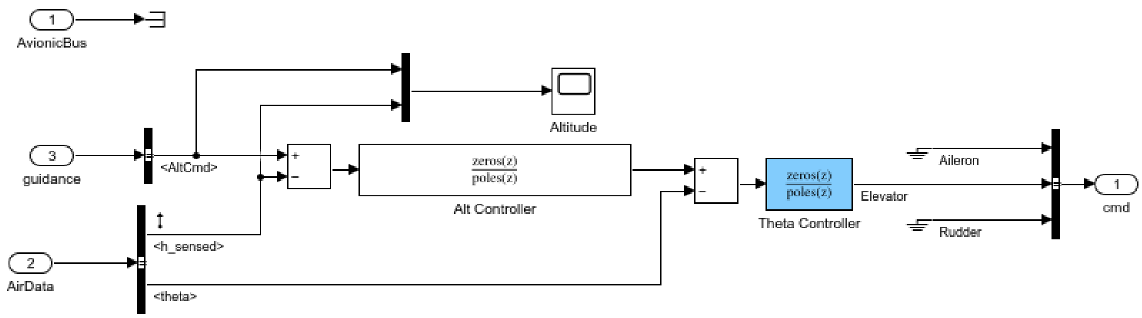


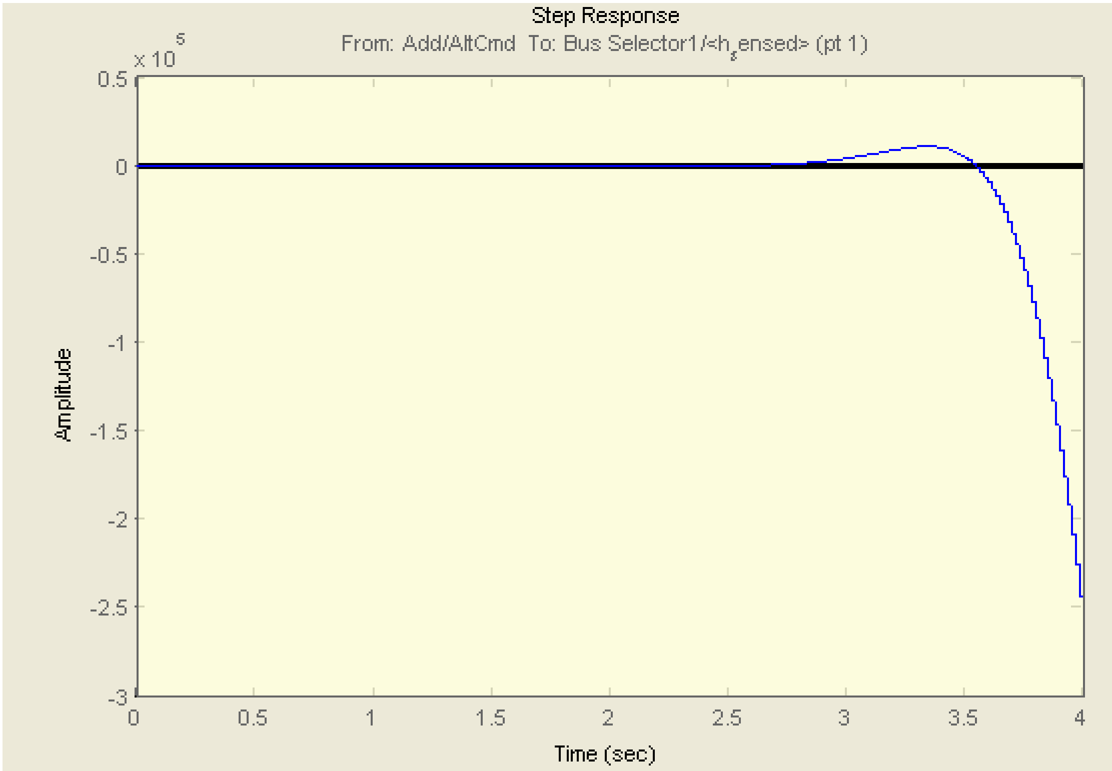
Figure 10: Longitudinal controller in Simulink model.

With Simulink Control Design™ software, you can tune the controllers directly in Simulink using a range of tools and techniques.

Using the Simulink Control Design interface, set up the control problem by specifying:

- Two controller blocks
- Closed-loop input or altitude command
- Closed-loop output signals or sensed altitude
- Steady-state or trim condition

Using this information, Simulink Control Design software automatically computes linear approximations of the model and identifies feedback loops to be used in the design. To design the controllers for the inner and outer loops, use root locus and bode plots for the open loops and a step response plot for the closed-loop response (Figure 11).


Step Response
From: Add/AltCmd To: Bus Selector1/<h_sensed> (pt 1)
x 10^5
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
Amplitude
0
0.5
1
1.5
2
2.5
3
3.5
4
Time (sec)

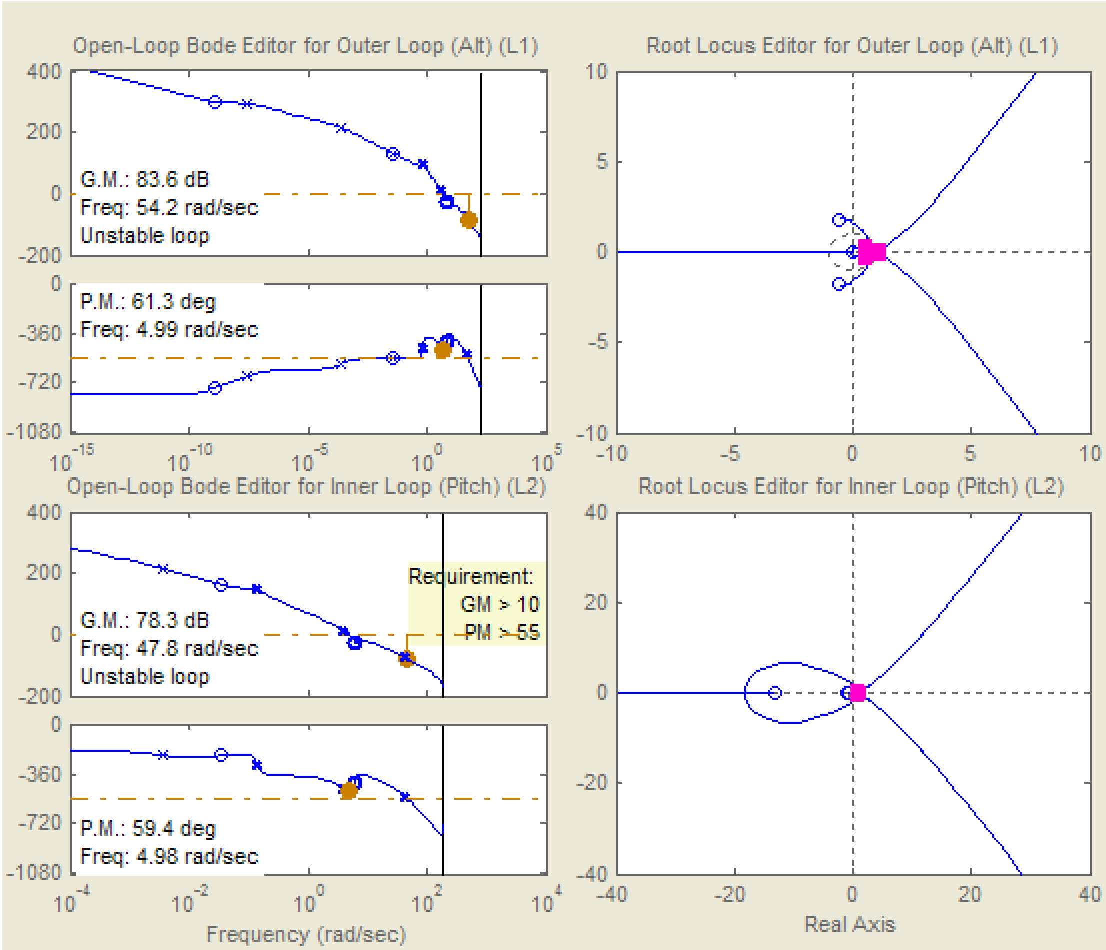


Figure 11: Design plots before controller tuning.

You then interactively tune the compensators for the inner and outer loops using these plots. Because the plots update in real time as you tune the compensators, you can see the coupling effects that these changes have on other loops and on the closed-loop response.

To make the multi-loop design more systematic, use a sequential loop closure technique. This technique lets us incrementally take into account the dynamics of the other loops during the design process. With Simulink Control Design, configure the inner loop to have an additional loop opening at the output of the outer loop controller (C1 in Figure 12). This approach decouples the inner loop from the outer loop and simplifies the inner-loop controller design. After designing the inner loop, design the outer loop controller. Figure 13 shows the resulting tuned compensator design at the final trimmed operating point.

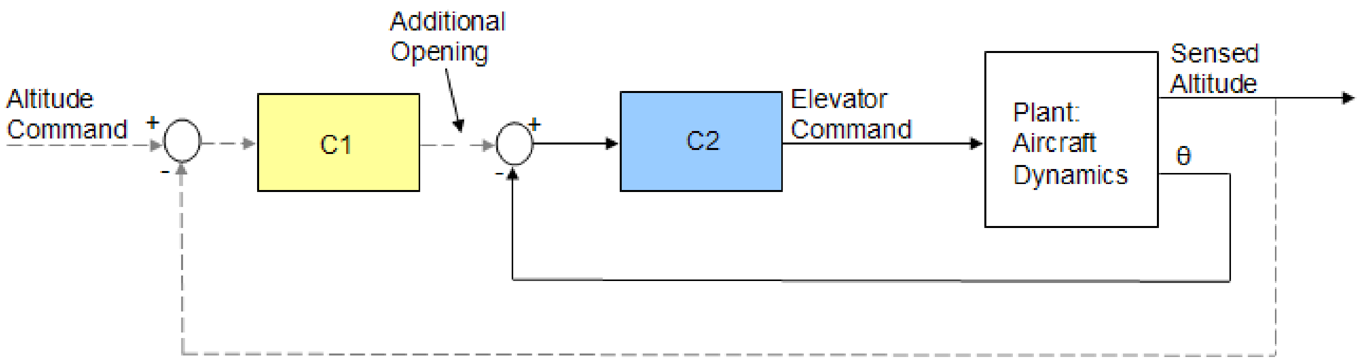
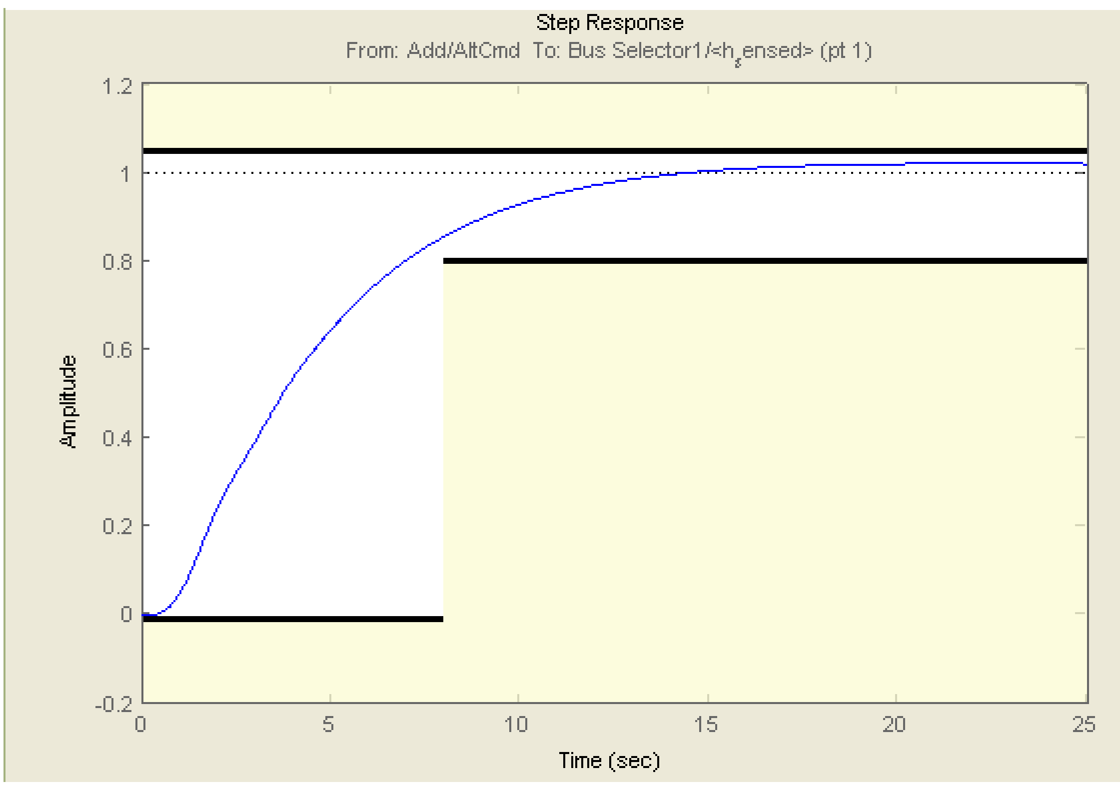


Figure 12: Block diagram of inner loop, isolated by configuring an additional loop opening.



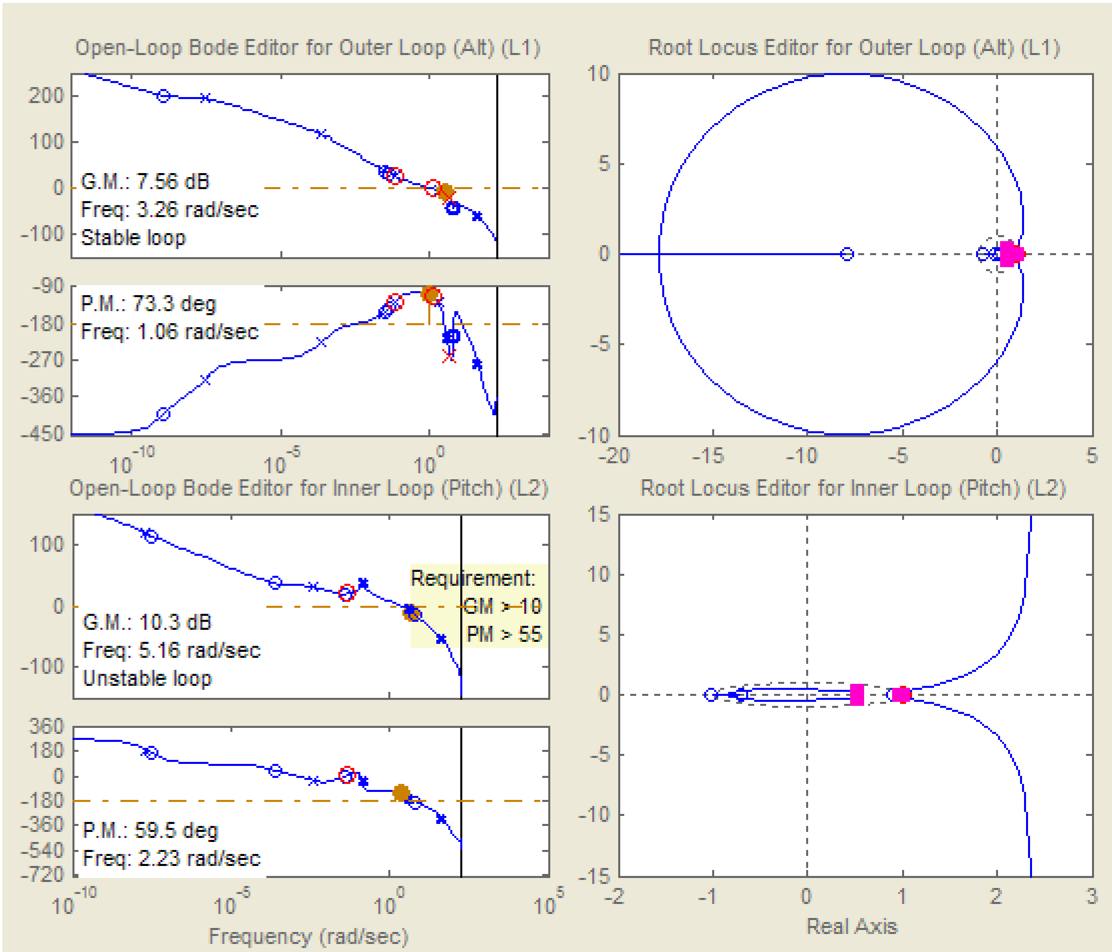


Figure 13: Design plots at trim condition after controller tuning.

You can tune the controller in Simulink Control Design software in several ways. For example:

- You can use a graphical approach, and interactively move controller gain, poles, and zeros until you get a satisfactory response (Figure 13).
- You can use Simulink Design Optimization™ software within Simulink Control Design software to tune the controller automatically.

After you specify frequency domain requirements, such as gain margin and phase margin and time domain requirements, Simulink Design Optimization software automatically tunes controller parameters to satisfy those requirements. Once you develope an acceptable controller design, the control blocks in the Simulink model are automatically updated. See the examples “Getting Started with the Control System Designer” (Control System Toolbox) in Control Systems Toolbox examples and “Tune Simulink Blocks Using Compensator Editor” (Simulink Control Design) in Simulink Control Design examples for more information on tuning controllers.

You can now run our nonlinear simulation with flight control logic and check that the controller performance is acceptable. Figure 15 shows the results from a closed-loop simulation of our nonlinear Simulink model for a requested altitude increase from 2000 meters to 2050 meters starting from a trimmed operating point. Although a pilot requests a step change in altitude, the actual controller altitude request rate is limited to provide a comfortable and safe ride for the passengers.

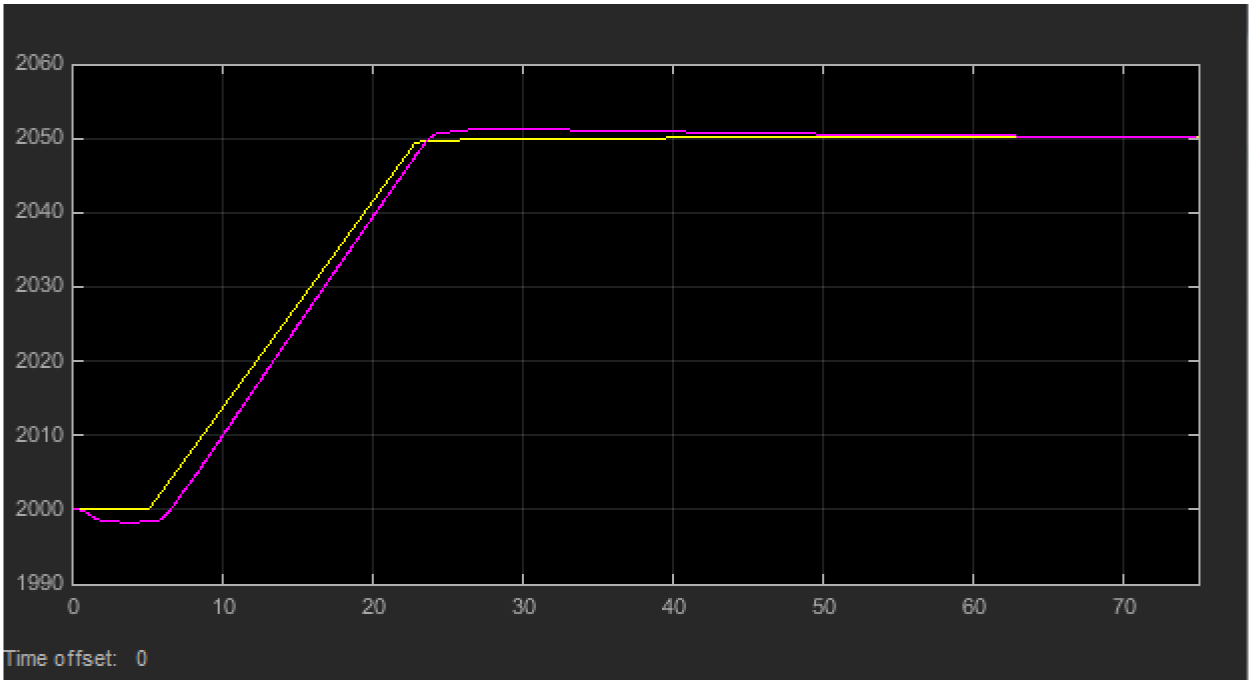


Figure 14: The final check is to run nonlinear simulation with our controller design and check that altitude (purple) tracks altitude request (yellow) in the stable and acceptable fashion.

You can now use these simulation results to determine whether our aircraft design meets its performance requirements. The requirement called for the climb rate to be above 2 m/s. As you can see, the aircraft climbed from 2000 to 2050 meters in less than 20 seconds, providing a climb rate higher than 2.5 m/s. Therefore, this particular geometric configuration and controller design meets our performance requirements.

In addition to traditional time plots, you can visualize simulation results using the Aerospace Blockset interface to FlightGear (Figure 15).

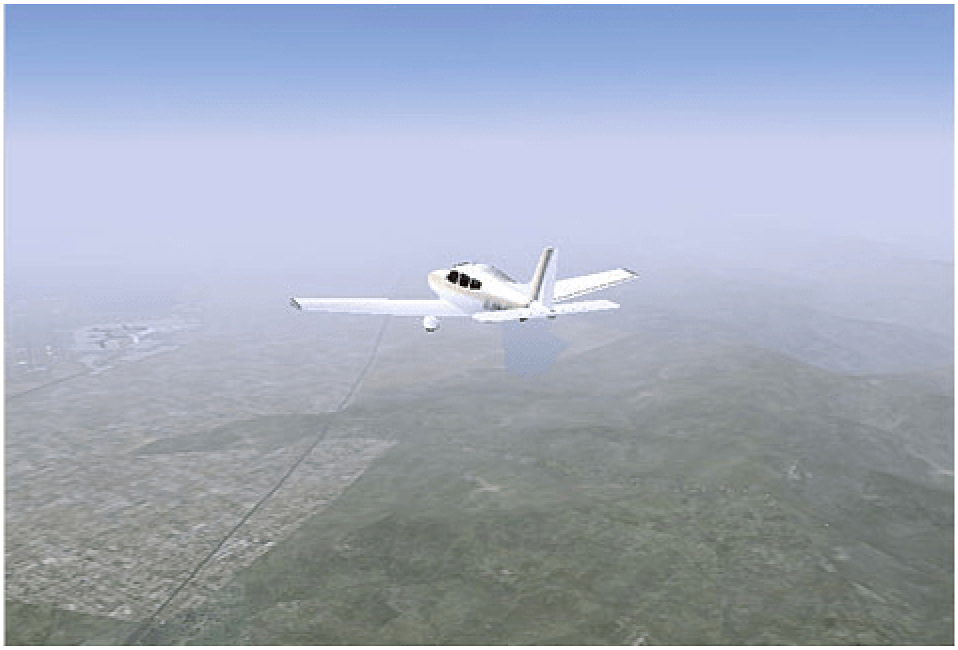


Figure 15: Visualizing simulation results using the Aerospace Blockset interface to FlightGear.

You can also use the Aerospace Toolbox interface to FlightGear to play back MATLAB data using either simulation results or actual flight test data.

Completing the Design Process

The next steps involve

1. Building a hardware-in-the-loop system to test real-time performance.
2. Building the actual vehicle hardware and software.
3. Conducting the flight test.
4. Analyzing and visualizing the flight test data.

Because these steps are not the focus of this example, they are not described here. Instead, it is simply mentioned that they can all be streamlined and simplified using the appropriate tools, such as Embedded Coder®, Simulink Real-Time™, and Aerospace Toolbox software.

Summary

This example showed how to:

- Use Digital Datcom and Aerospace Toolbox software to rapidly develop the initial design of your flight vehicle and evaluate different geometric configurations.
- Use Simulink and Aerospace Blockset software to rapidly create a flight simulation of your vehicle.
- Use Simulink Control Design software to design flight control laws.

This approach enables you to determine the optimal geometrical configuration of your vehicle and estimate its performance and handling qualities well before any hardware is built, reducing design costs and eliminating errors. In addition, using a single tool chain helps facilitate communication among different groups and accelerates design time.

References

[1] Cannon, M, Gabbard, M, Meyer, T, Morrison, S, Skocik, M, Woods, D. "Swineworks D-200 Sky Hogg Design Proposal." AIAA®/General Dynamics Corporation Team Aircraft Design Competition, 1991-1992.

[2] Turvesky, A., Gage, S., and Buhr, C., "Accelerating Flight Vehicle Design", MATLAB Digest, January 2007.

[3] Turvesky, A., Gage, S., and Buhr, C., "Model-based Design of a New Lightweight Aircraft", AIAA paper 2007-6371, AIAA Modeling and Simulation Technologies Conference and Exhibit, Hilton Head, South Carolina, Aug. 20-23, 2007.

See Also

Digital DATCOM Forces and Moments | Aerodynamic Forces and Moments

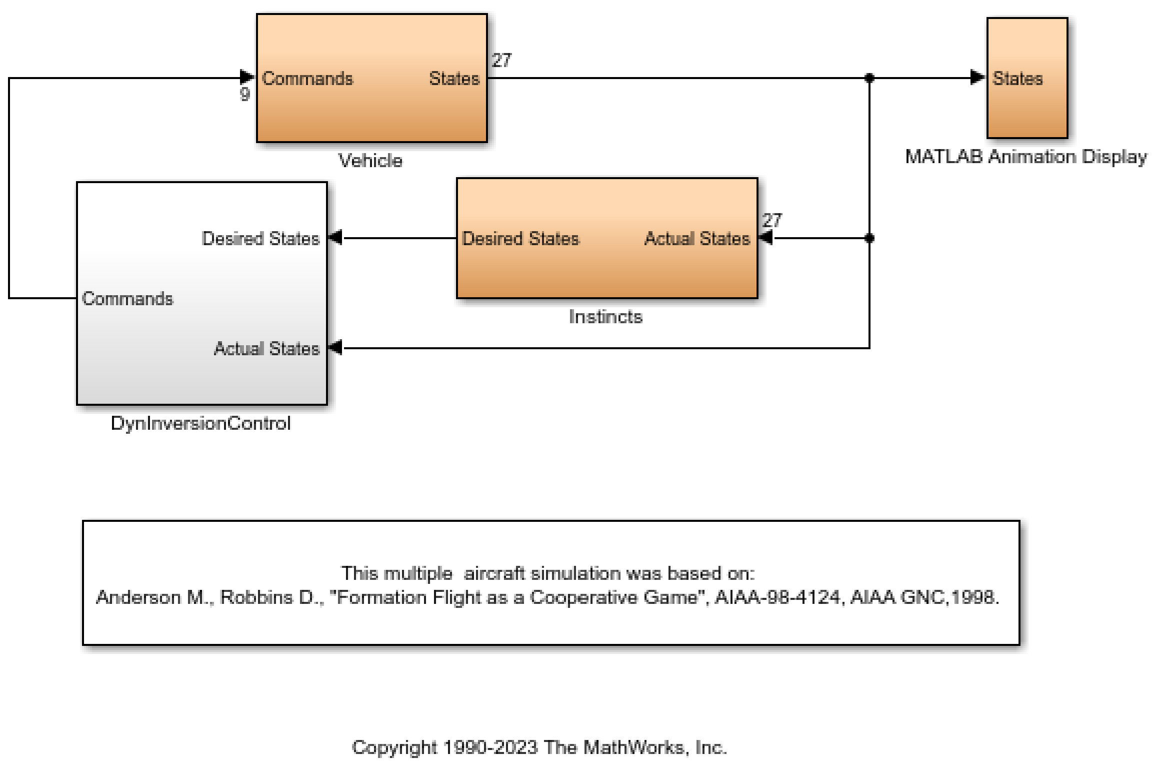
Multiple Aircraft with Collaborative Control

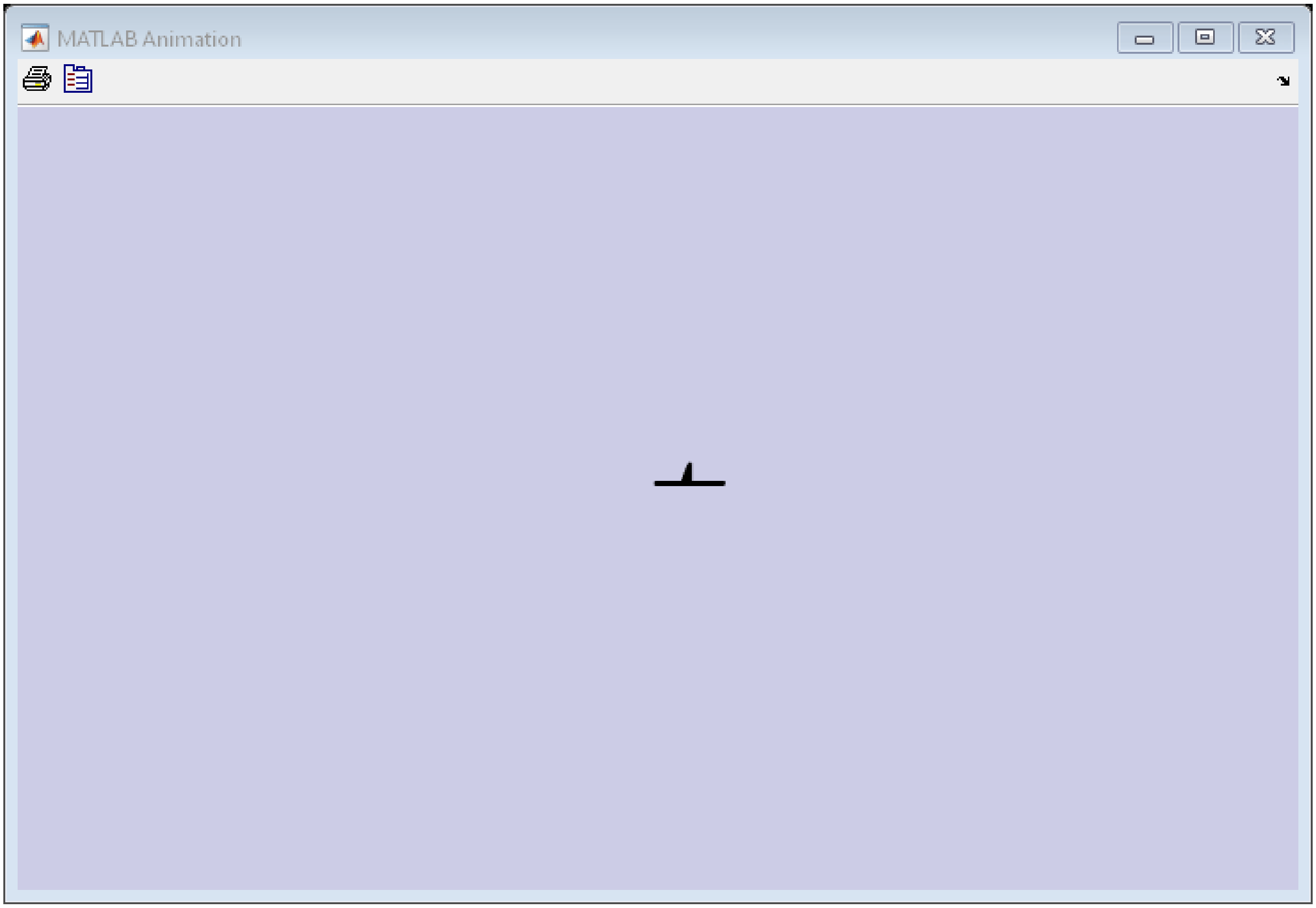
This example shows the simulation of multiple aircraft in formation flight. For easy updates and the specification of arbitrary numbers of vehicles, the example is vectorized. To perform their avoidance task, this set of aircraft uses cooperative control.

To help identify Aerospace Blockset™ blocks, this model uses color coding. The red blocks are Aerospace Blockset blocks, the orange blocks are subsystems that contain additional Aerospace Blockset blocks, and the white blocks are Simulink® blocks.

The simulation uses Simulink and Aerospace Blockset software, which allow for a hierarchal block diagram representation to include the control laws, vehicle models, and visualization.

```
open_system('asbswarm');
```



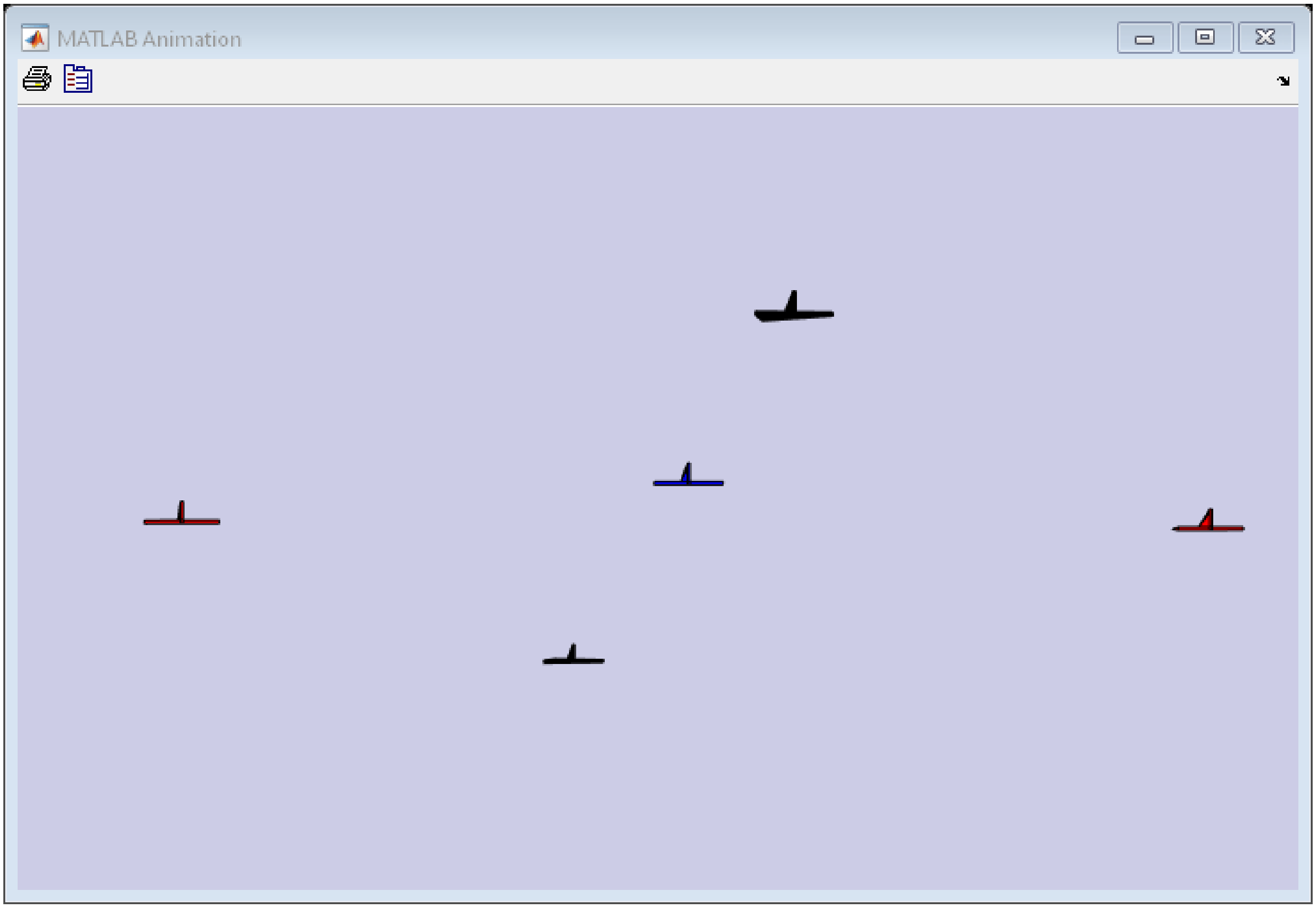


The MATLAB® Animation Display subsystem contains the MATLAB® Animation block from Aerospace Blockset to visualize the simulation.

There are three types of bodies:

- A blue body is first in the formation. It is the target of the camera.
- Red bodies are the second and third bodies in the formation.
- Two black bodies represent the obstacles.

```
sim('asbswarm','StopTime','10');
```



The basis of this simulation comes from previous research performed in the study of aircraft formation flight in the context of cooperative game theory and the natural aggregate motion of flocking birds, schooling fish, and the herding of land animals.

See Also

6th Order Point Mass (Coordinated Flight) | MATLAB Animation

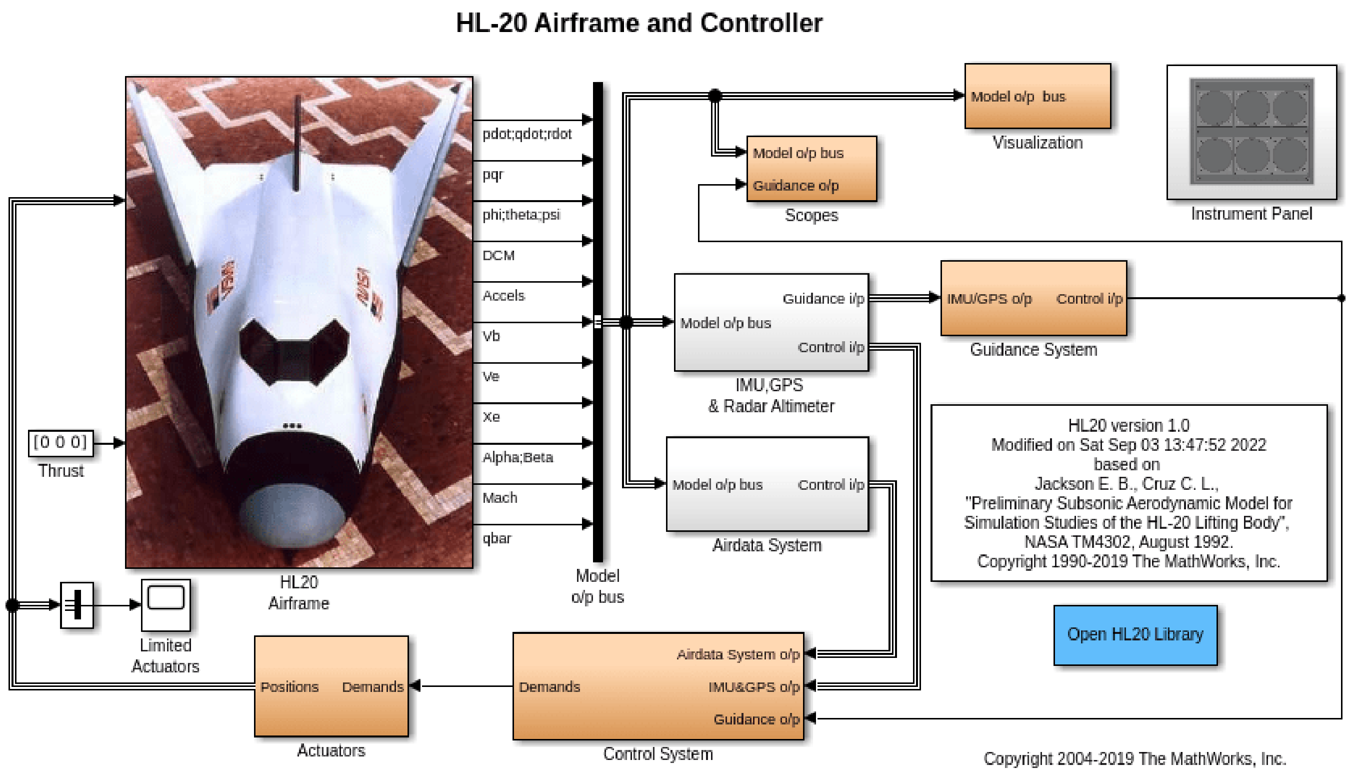
HL-20 with Flight Instrumentation Blocks

This example shows NASA's HL-20 lifting body and controller modelled in Simulink® and Aerospace Blockset™ software. This example model simulates approach and landing flight phases using an auto-landing controller. The Visualization subsystem uses aircraft-specific gauges from the Aerospace Blockset Flight Instrumentation library.

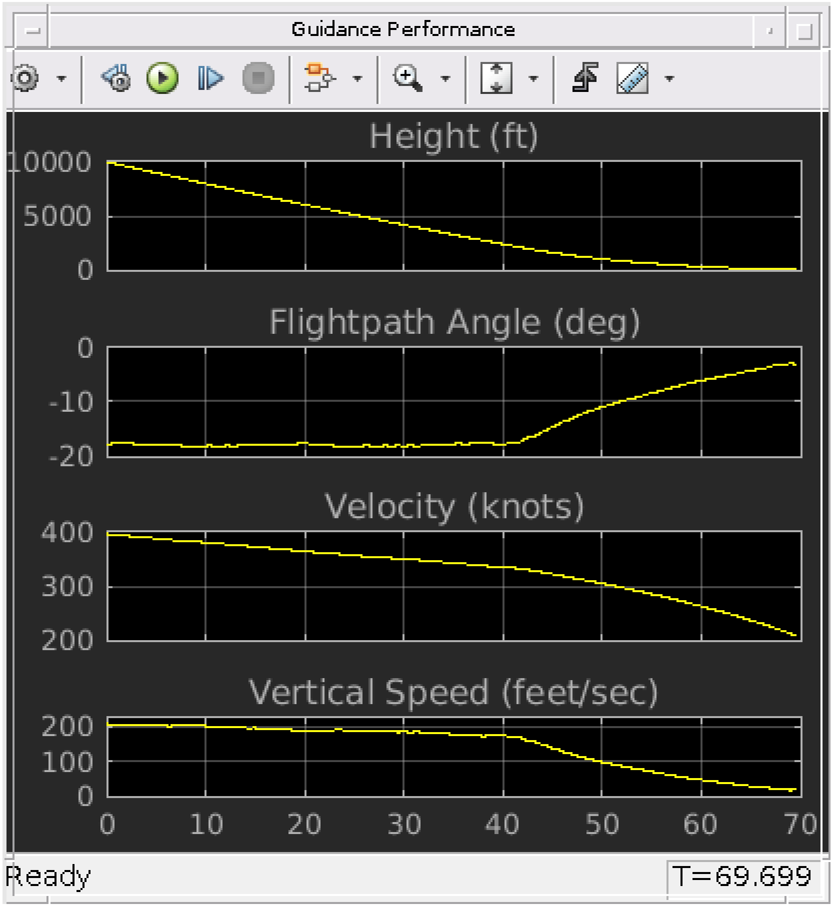
The HL-20, also known as personnel launch system (PLS), is a lifting body re-entry vehicle designed to complement the Space Shuttle orbiter. Designed to carry up to ten people and very little cargo[1], the HL-20 lifting body was to be placed in orbit either launched vertically via booster rockets or transported in the payload bay of the Space Shuttle orbiter. The HL-20 lifting body was designed to have powered deorbiting accomplished with an onboard propulsion system, while its reentry was to be nose-first, horizontal, and unpowered.

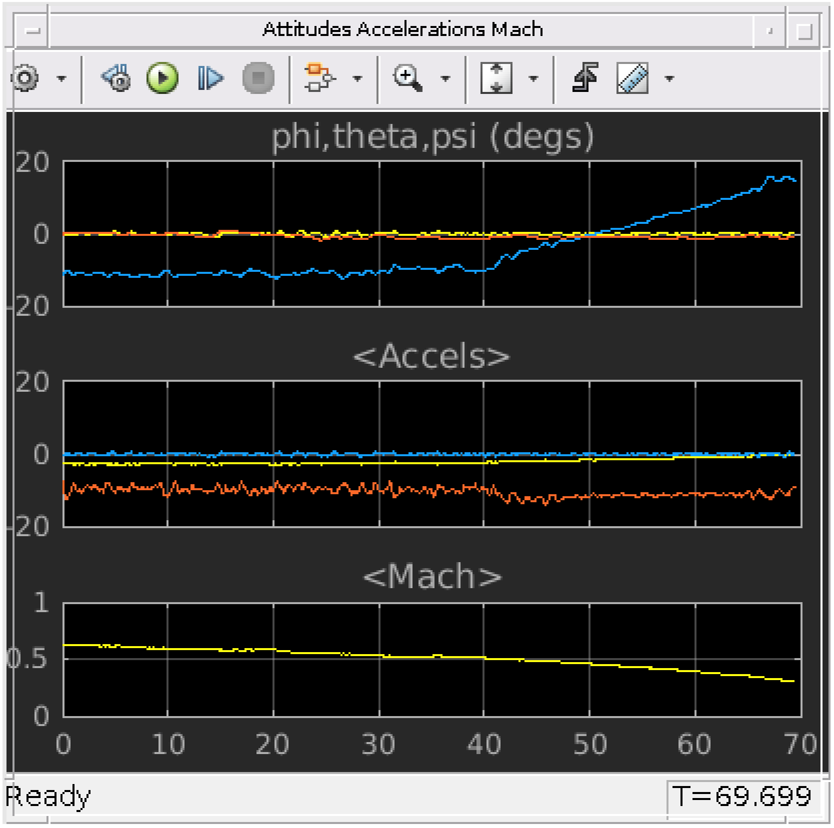
The HL-20 lifting body was developed as a low cost solution for getting to and from low Earth orbit. The proposed benefits of the HL-20 were reduced operating costs due to rapid turnaround between landing and launch, improved flight safety, and ability to land conventionally on runways. Potential scenarios for the HL-20 were orbital rescue of stranded astronauts, International Space Station crew exchange if the Space Shuttle orbiter was not available, observation missions, and satellite servicing missions.

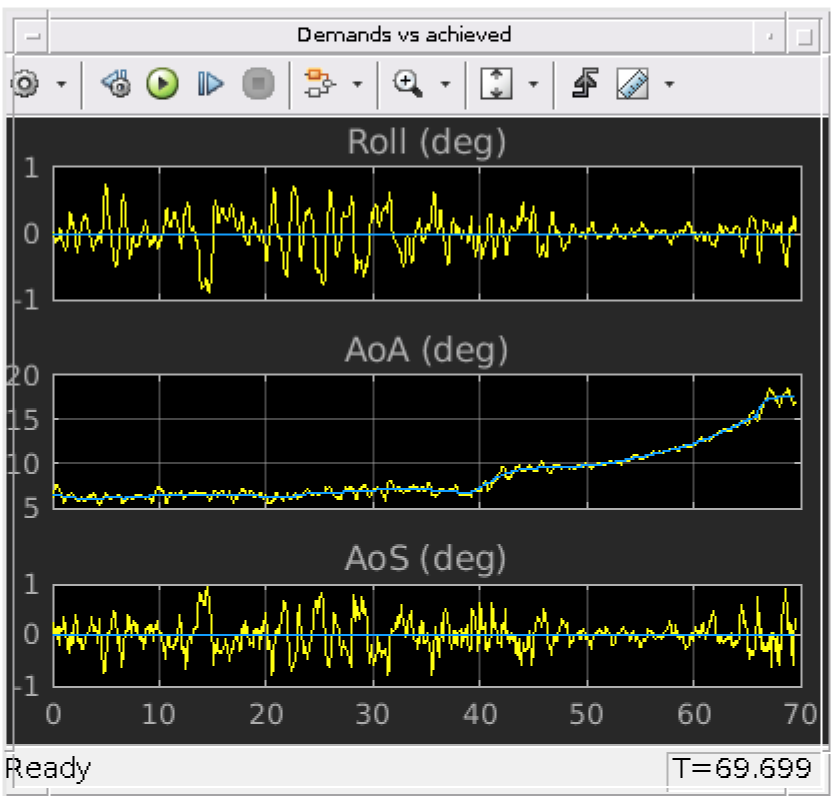
```
mdl = "HL20Gauges";
open_system(mdl);
```

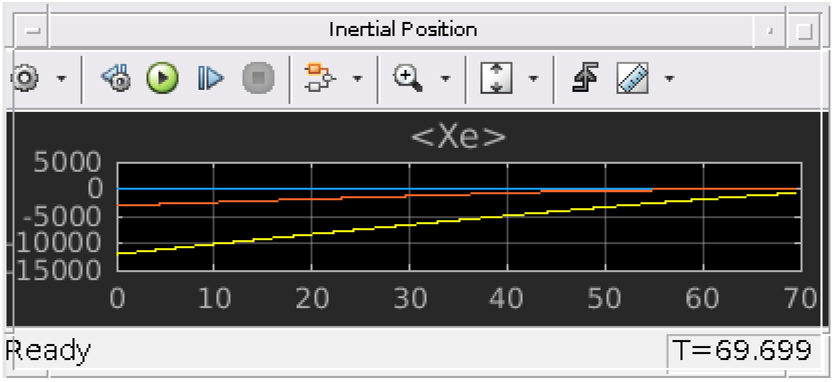



450
400
0
AIRSPEED
350
50
300
100
KNOTS
250
150
200
40
30
20
10
-10
100 FEET
9
1
8
2
7
3
6
4
5
D.C.
ELEC.
TURN COORDINATOR
L
R
NO PITCH
INFORMATION
N
33
3
30
6
W
E
24
12
21
15
S
VERTICAL SPEED
1000FT PER MIN
UP
DOWN
0
4
8
12
16


Guidance Performance
Height (ft)
10000
5000
0
Flightpath Angle (deg)
0
-10
-20
Velocity (knots)
400
300
200
Vertical Speed (feet/sec)
200
100
0
0
10
20
30
40
50
60
70
Ready
T=69.699


Attitudes Accelerations Mach
phi,theta,psi (degs)
20
0
-20
<Accels>
20
0
-20
<Mach>
1
0.5
0
0 10 20 30 40 50 60 70
Ready
T=69.699


Demands vs achieved
Roll (deg)
1
0
-1
AoA (deg)
20
15
10
5
AoS (deg)
1
0
-1
0 10 20 30 40 50 60 70
Ready
T=69.699



Additional information about HL-20

[1] Jackson E. B., Cruz C. L., "Preliminary Subsonic Aerodynamic Model for Simulation Studies of the HL-20 Lifting Body," NASA TM4302 (August 1992)

See Also

Related Examples

- “NASA HL-20 Lifting Body Airframe” on page 3-14
- “Flight Instrument Gauges” on page 2-58

HL-20 with Flight Instrument Blocks and Visualization Using Unreal Engine

This model shows NASA's HL-20 lifting body and controller modeled in Simulink® and Aerospace Blockset™, using Unreal Engine® for visualization. This model simulates approach and landing flight phases using an auto-landing controller. The Instrument Panel subsystem uses aircraft-specific gauges from the Aerospace Blockset™ Flight Instrumentation library.

The HL-20, also known as personnel launch system (PLS), is a lifting body reentry vehicle designed to complement the Space Shuttle orbiter. Designed to carry up to ten people and very little cargo[1], the HL-20 lifting body was to be placed in orbit either launched vertically via booster rockets or transported in the payload bay of the Space Shuttle orbiter. HL-20 lifting body was designed to have a powered deorbiting accomplished with an onboard propulsion system while its reentry was to be nose-first, horizontal, and unpowered.

The HL-20 lifting body was developed as a low cost solution for getting to and from low Earth orbit. The proposed benefits of the HL-20 were reduced operating costs due to rapid turnaround between landing and launch, improved flight safety, and ability to land conventionally on runways. Potential scenarios for the HL-20 were orbital rescue of stranded astronauts, International Space Station crew exchange if the Space Shuttle orbiter was not available, observation missions, and satellite servicing missions.

Note: This example is not supported in Simulink Online.

Create HL-20 Skeletal Mesh

To prepare a new aircraft for use in Aerospace Blockset™ with Unreal Engine® (UE) visualization, create a skeletal mesh that uses one of the custom skeletons. This was done for HL-20 using the Custom aircraft skeleton and the following workflow.

1. Get the AC3D file from ~/matlab/toolbox/aero/animation/HL20/Models/HL20.ac.
2. If using Blender for the conversion, set it up to import AC3D files (cf. https://wiki.flightgear.org/Howto:Work_with_AC3D_files_in_Blender).
3. Follow the seven steps in “Prepare Custom Aircraft Mesh for the Unreal Editor” on page 4-34 to convert file “HL20.ac” to “HL20.fbx”, and then import that skeletal mesh into UE.

For HL-20, many of the bones are unnecessary and therefore are not connected to any mesh objects. For example, there is no second wing or any propellers. However, you must still create all the bones so that its skeleton matches the Custom skeleton in the support package.

The HL-20 model in Aerospace Blockset has the following skeletal bone connections.

Bone Index:	Bone Name:	HL-20 Mesh Part:
1	FixedWing	Entire aircraft
2	Engine1	Left main engine
4	Engine2	Right main engine
6	Engine3	Left rear thruster group
8	Engine4	Right rear thruster group
34	Wing1	Wing
35	Wing1_Flap_L	Lower left body flap
36	Wing1_Flap_R	Lower right body flap
37	Wing1_Aileron_L	Left wing flap (acts as aileron and elevator)
38	Wing1_Aileron_R	Right wing flap (acts as aileron and elevator)
39	Wing1_Spoiler_L	Upper left body flap
40	Wing1_Spoiler_R	Upper right body flap
47	Rudder_L	All-moving vertical stabilizer
49	NoseGear	Nose gear
50	NoseGear_Wheel	Nose gear wheel
52	MainGear_L	Left main gear
53	MainGear_L_Wheel	Left main gear wheel
55	MainGear_R	Right main gear
56	MainGear_R_Wheel	Right main gear wheel

Use Unreal Engine Visualization

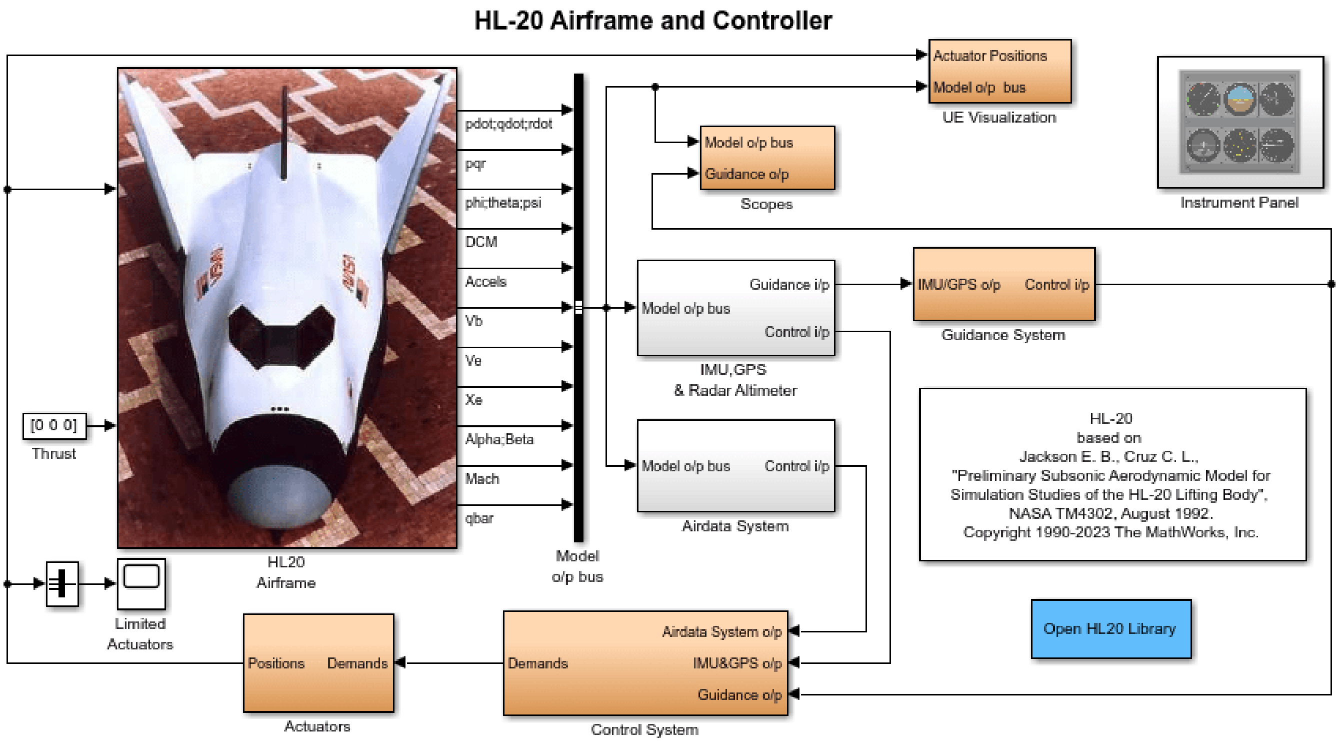
Once the custom mesh is available as an aerospace asset for UE, the Aerospace Blockset > Animation > Simulation 3D > Simulation 3D Aircraft block can use it to implement the HL-20 airframe.

NASA HL-20 Lifting Body Airframe Example

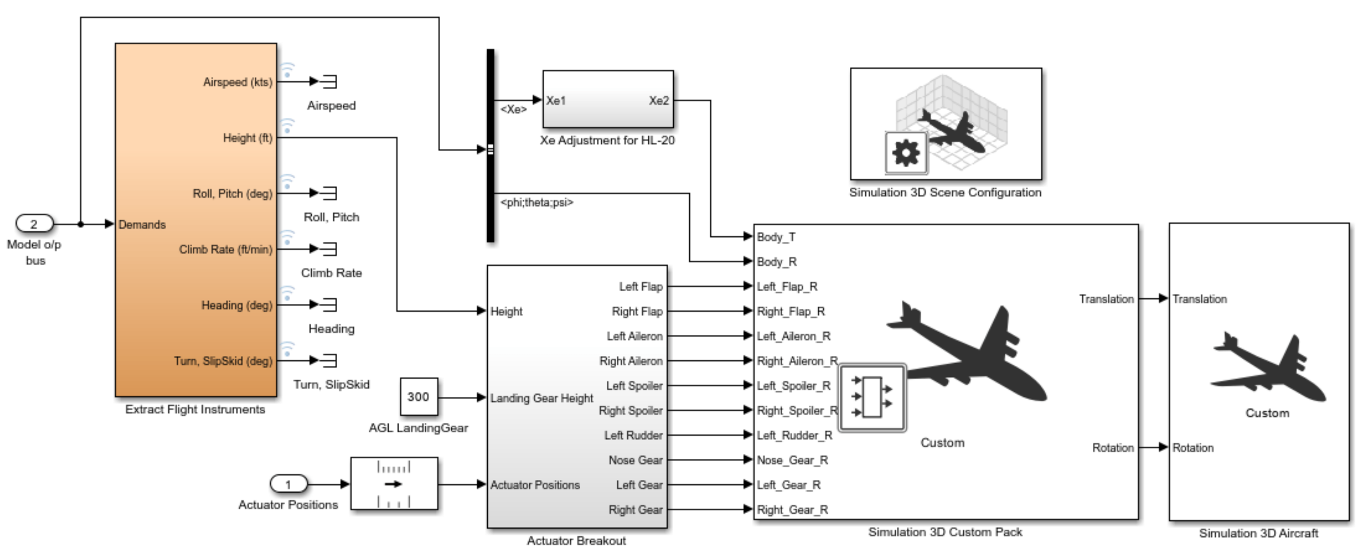
Open the example model.

```
mdl = "aeroblk_HL20_UE";
open_system(mdl);
```

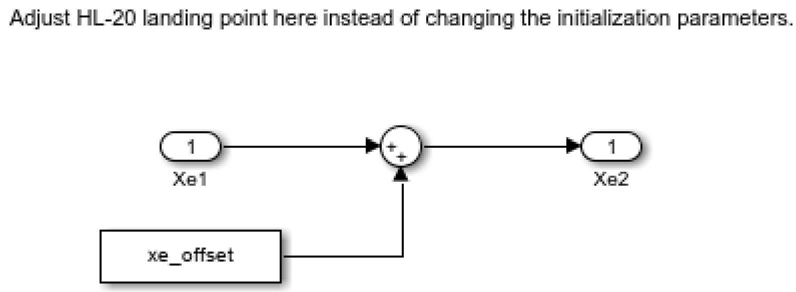
The model appears as follows.



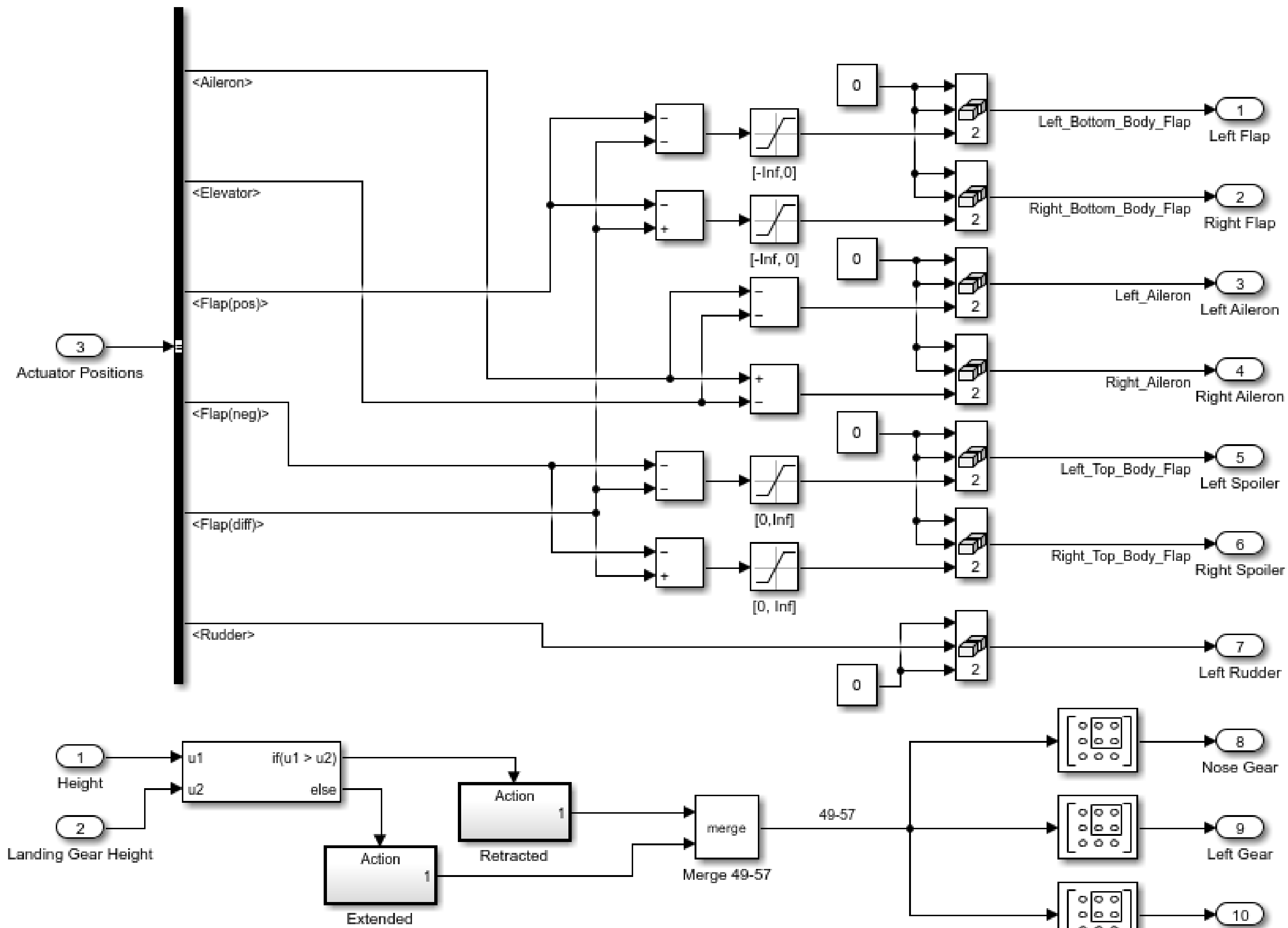
The UE Visualization subsystem (shown below) sends the body location (Xe) and rotation (`[phi, theta, psi]`) to the Simulation 3D Custom Pack block, which assembles the translation and rotation matrices for the Simulation 3D Aircraft block to position the HL-20 model and all of its parts. For more realism, this model adds the rotations of the control surfaces as well as landing gear retraction and extension. These rotations are calculated in the Actuator Breakout subsystem.



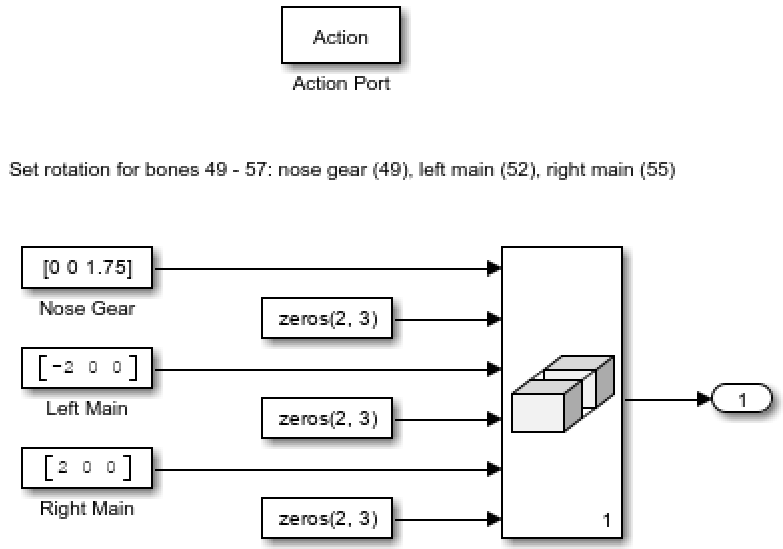
To place the HL-20 correctly in this airport scene, the Xe Adjustment for HL-20 subsystem adds an offset to the incoming location.



The Actuator Breakout subsystem (shown below) sets the position for all the control surfaces as well as for the landing gear. The landing gear is initialized in the retracted state and set to extend when the height above ground level (AGL) is below a certain number (e.g. 300 feet). The actuator commands cannot all be directly applied to the control surfaces, since the HL-20 is an unusual aircraft with high-dihedral wings with single flaps, no tail, and four body flaps, two on the top and two on the bottom. The wing flaps are used for both aileron and elevator actions, whereas the body flaps are used as conventional flaps, spoilers, and in differential mode to assist the ailerons.

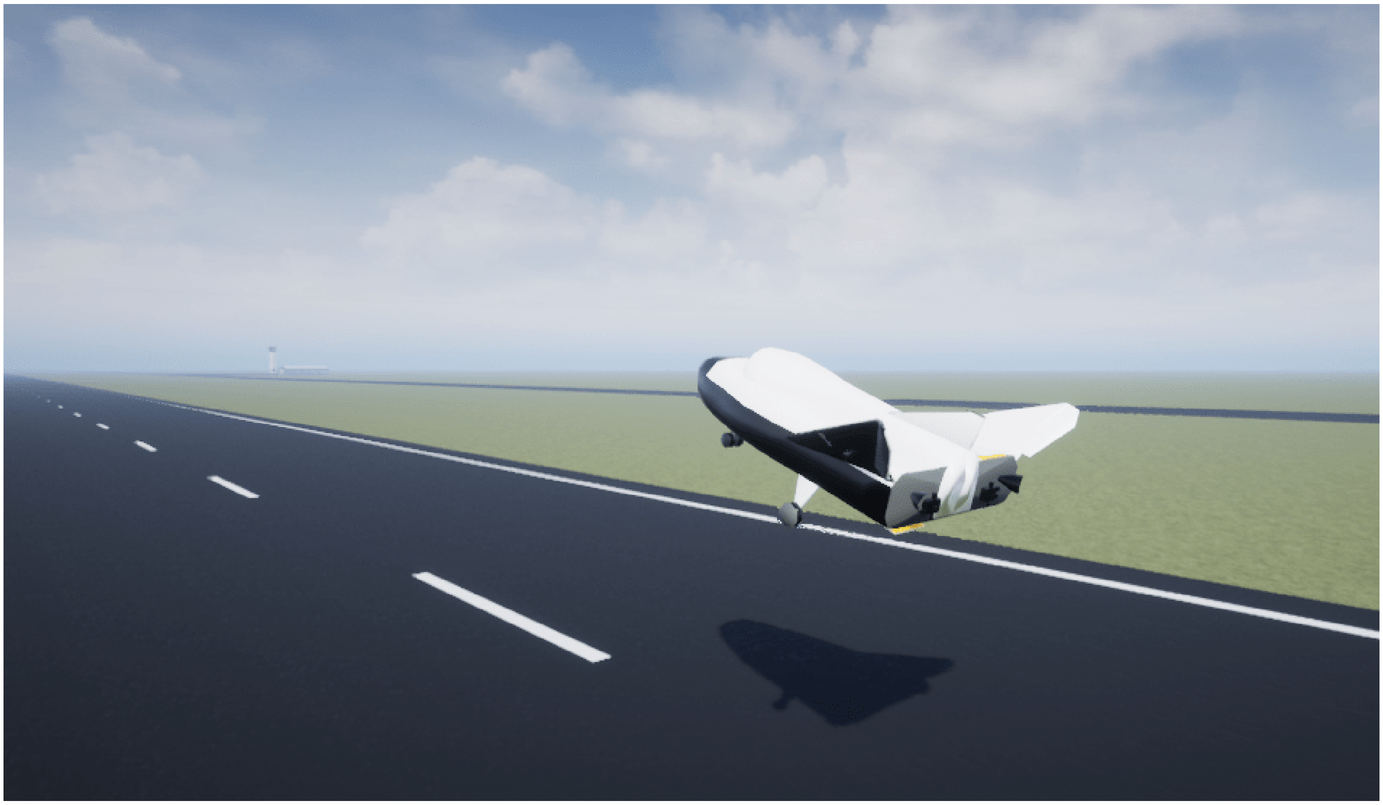


The Retracted Action subsystem (shown below) for the landing gear rotates them to their retracted positions, whereas the Extended Action subsystem returns all rotations to zero.



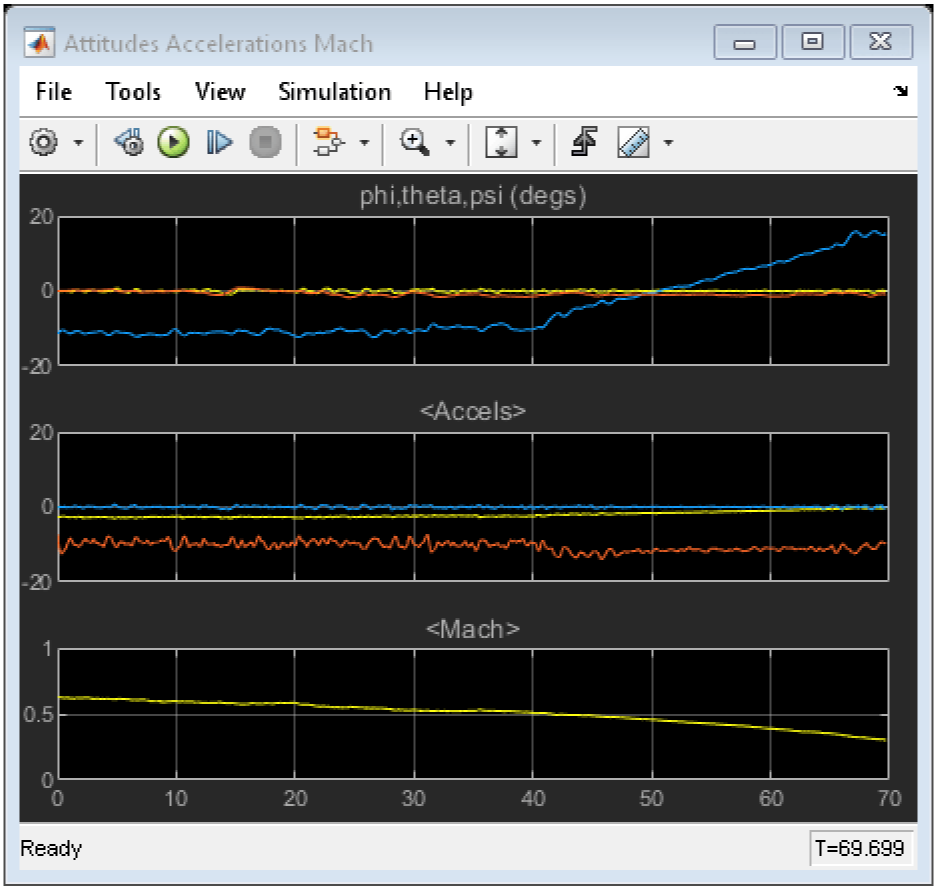
Before running the model, note that **Simulation Pacing** has been turned on so that the simulation clock matches the rate at which the model data was taken.

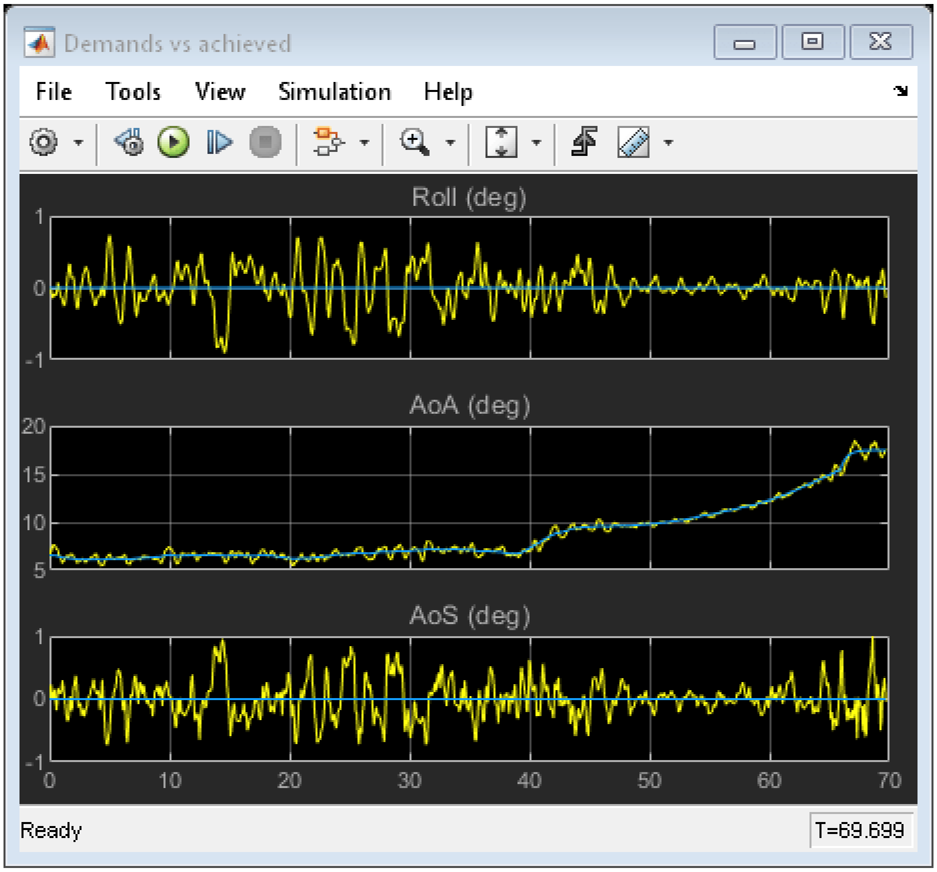
- After clicking the **Run** button, allow a few seconds for the 3D visualization window to initialize.
- You should now see the HL-20 descending towards the airport, initially with the landing gear retracted.
- Once it is simulating, you can switch between camera views by first left-clicking inside the 3D window, then using the keys `0` through `9` to choose between ten preconfigured camera positions. For more information on camera views, see the **Run Simulation** section in **Customize Scenes Using Simulink and Unreal Editor**.

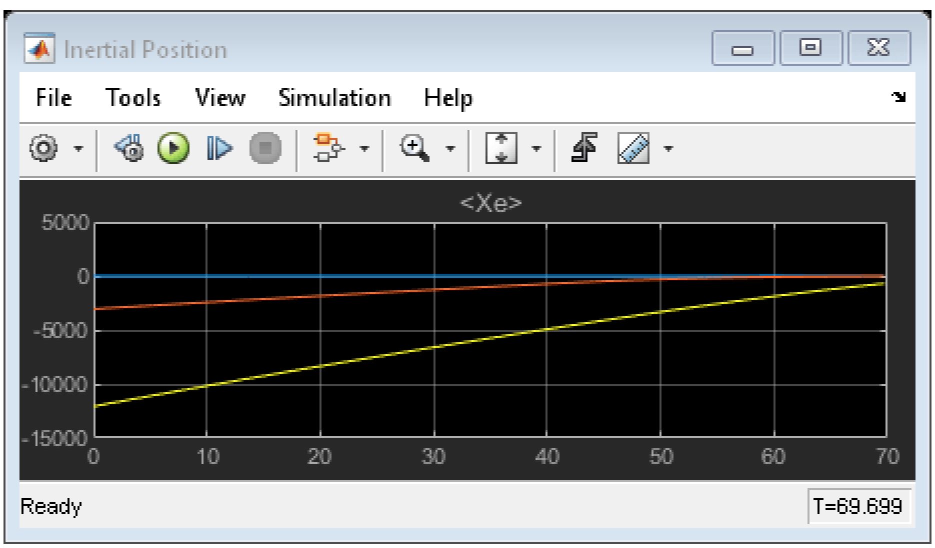


To see the flight instrument gauges during the run, open the Instrument Panel.


AIRSPEED
KNOTS
VERTICAL SPEED
1000FT PER MIN
UP
DOWN
D.C.
ELEC.
TURN COORDINATOR
NO PITCH
INFORMATION
L
R


Attitudes Accelerations Mach
File Tools View Simulation Help
phi,theta,psi (degs)
<Accels>
<Mach>
Ready
T=69.699


Demands vs achieved
File Tools View Simulation Help
Roll (deg)
1
0
-1
AoA (deg)
20
15
10
5
AoS (deg)
1
0
-1
0 10 20 30 40 50 60 70
Ready
T=69.699


Inertial Position
File Tools View Simulation Help
<Xe>
5000
0
-5000
-10000
-15000
0 10 20 30 40 50 60 70
Ready
T=69.699

References

[1] Jackson, E. B., and C. L. Cruz, “Preliminary Subsonic Aerodynamic Model for Simulation Studies of the HL-20 Lifting Body,” NASA TM4302 (August 1992).

See Also
Simulation 3D Aircraft

Related Examples
- “NASA HL-20 Lifting Body Airframe” on page 3-14
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-37
- “How 3D Simulation for Aerospace Blockset Works” on page 2-40
- “Flight Simulator Interface” on page 2-20

HL-20 Project with Optional FlightGear Interface

This project shows how to model NASA's HL-20 lifting body with Simulink®, Stateflow® and Aerospace Blockset™ software. The vehicle model includes the aerodynamics, control logic, fault management systems (FDIR), and engine controls (FADEC). It also includes effects of the environment, such as wind profiles for the landing phase. The entire model simulates approach and landing flight phases using an auto-landing controller. To analyze the effects of actuator failures and wind gust variation on the stability of the vehicle, use the "Run Failure Analysis in Parallel" project shortcut. If Parallel Computing Toolbox™ is installed, the analysis is run in parallel. If Parallel Computing Toolbox™ is not installed, the analysis is run in serial. Visualization for this model is done via an interface to FlightGear, an open source flight simulator package. If the FlightGear interface is unavailable, you can simulate the model by closing the loop using the alternative data sources provided in the Variant block. In this block, you can choose a previously saved data file, a Signal Editor block, or a set of constant values. This example requires Control System Toolbox™.

Note: This example is not supported in Simulink Online.

FlightGear Interface

For more information on the FlightGear interface, read these documentation topics:

- “Flight Simulator Interface” on page 2-20
- “Work with the Flight Simulator Interface” on page 2-24
- “Run the HL-20 Example with FlightGear” on page 2-32

For a more detailed description of this model components, view a recorded navigation through the model using this link:

- Spacecraft Automated Landing System

NASA HL-20 Background

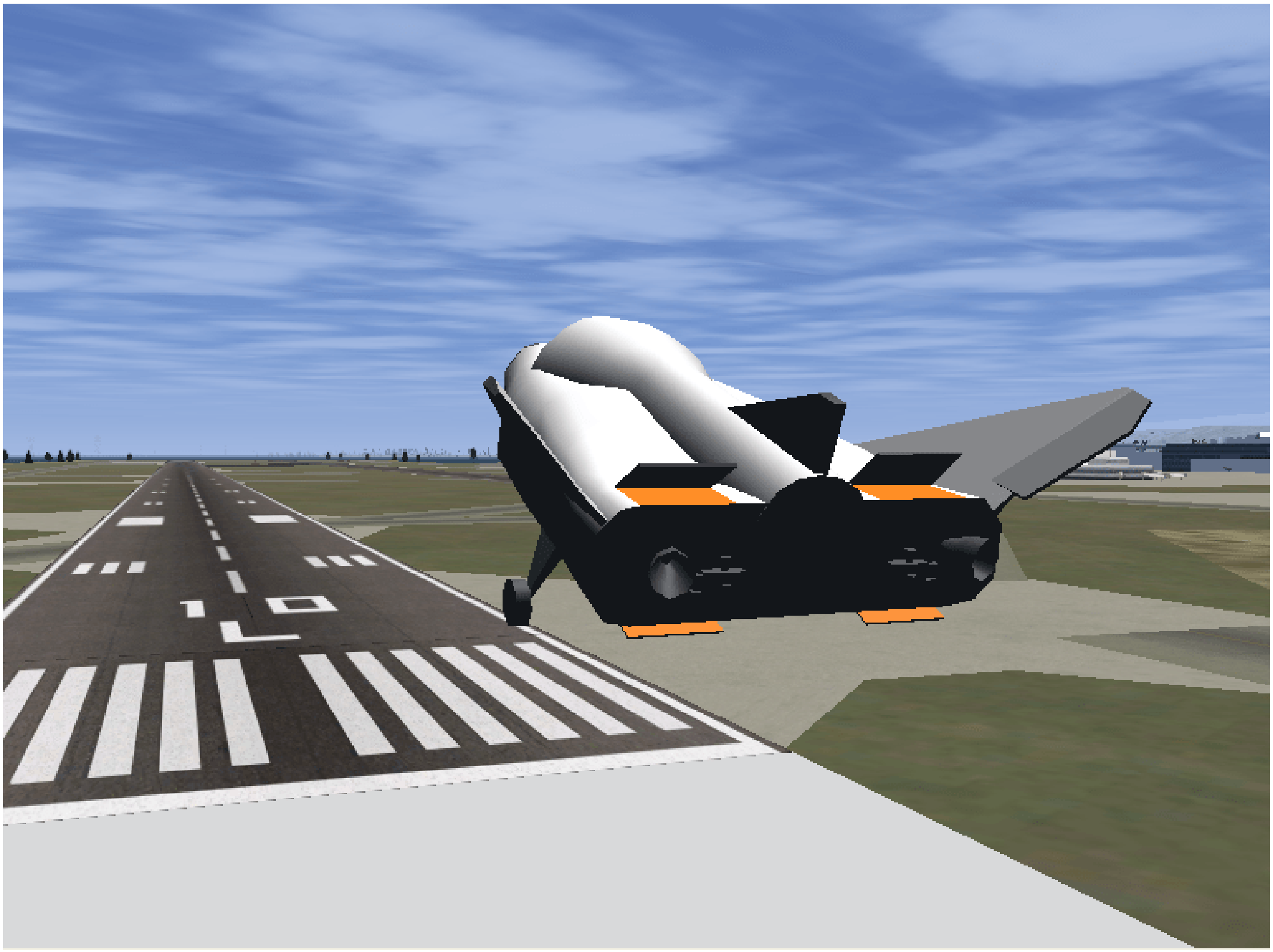
The HL-20, also known as personnel launch system (PLS), is a lifting body re-entry vehicle that was designed to complement the Space Shuttle orbiter. Designed to carry up to ten people and very little cargo[1], the HL-20 lifting body was to be placed in orbit either launched vertically via booster rockets or transported in the payload bay of the Space Shuttle orbiter. HL-20 lifting body was designed to have a powered deorbiting accomplished with an onboard propulsion system while its reentry was to be nose-first, horizontal and unpowered.

The HL-20 lifting body was developed as a low cost solution for getting to and from low Earth orbit. The proposed benefits of the HL-20 were reduced operating costs due to rapid turnaround between landing and launch, improved flight safety, and ability to land conventionally on runways. Potential scenarios for the HL-20 were orbital rescue of stranded astronauts, International Space Station crew exchange if the Space Shuttle orbiter was not available, observation missions, and satellite servicing missions.

Opening HL-20 Project

Run the following command to create and open a working copy of the project files for this example.

```
openProject('asbhl20');
```



References

[1] Tamayo, Sergio, Stacey Gage, and Gavin Walker. “Integrated Project Management Tool for Modeling and Simulation of Complex Systems.” *AIAA Modeling and Simulation Technologies Conference*, 2012. https://doi.org/10.2514/6.2012-4714.

[2] Jackson, E. Bruce, and Christopher I. Cruz. “Preliminary Subsonic Aerodynamic Model for Simulation Studies of the HL-20 Lifting Body.” *NTRS - NASA Technical Reports Server*. NASA-TM-4302, August 1, 1992. https://ntrs.nasa.gov/citations/19920021931.

See Also

Related Examples

- “NASA HL-20 Lifting Body Airframe” on page 3-14
- “Flight Simulator Interface” on page 2-20
- Spacecraft Automated Landing System

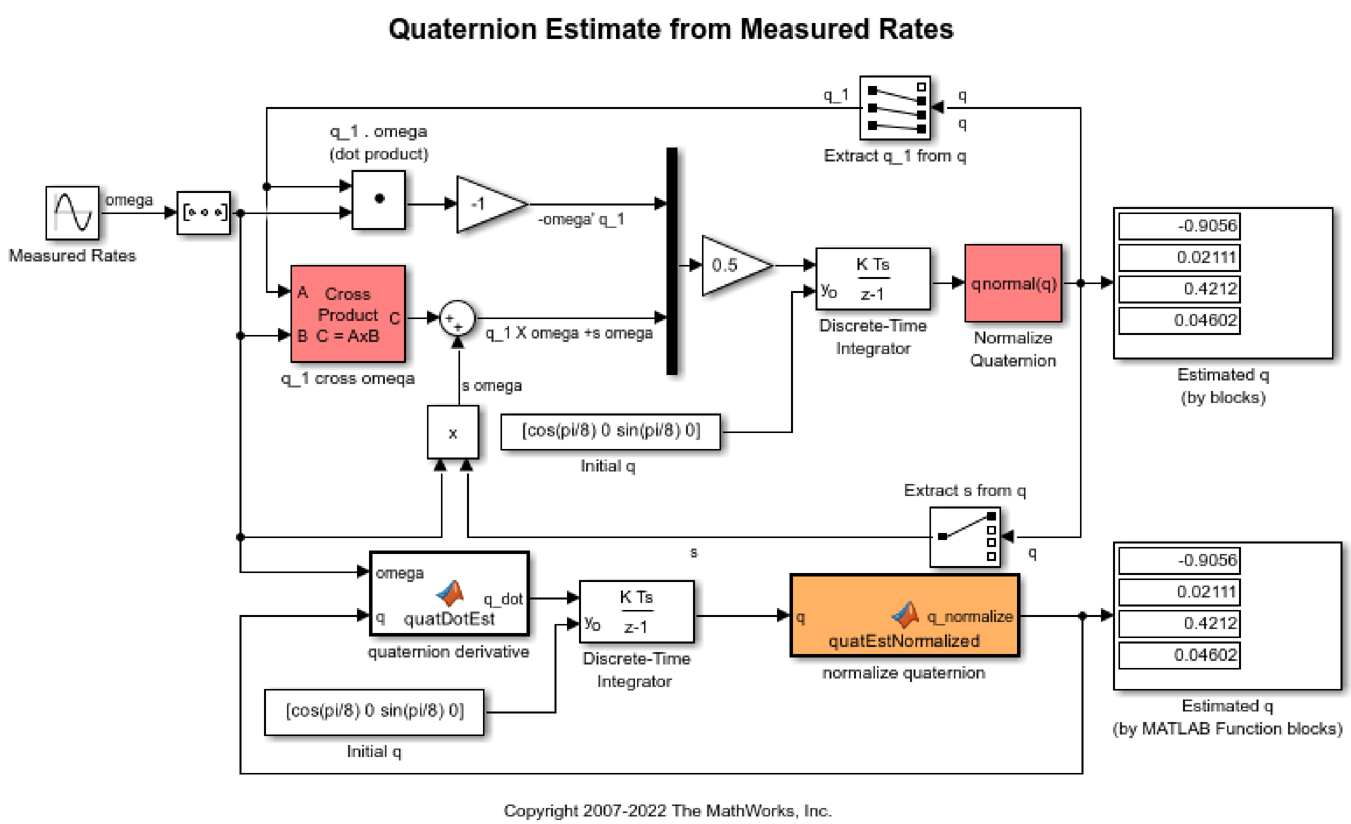
Quaternion Estimate from Measured Rates

This example shows how to estimate a quaternion and model the equations in the following ways:

- Using Simulink® and Aerospace Blockset™ software to implement the equations.
- Using MATLAB® Function block to incorporate an Aerospace Toolbox quaternion function.

This model has been color coded to aid in locating Aerospace Blockset blocks.

The red blocks are Aerospace Blockset blocks, the orange block is a MATLAB Function block containing a function with MATLAB function block support provided by Aerospace Blockset and the white blocks are Simulink blocks.



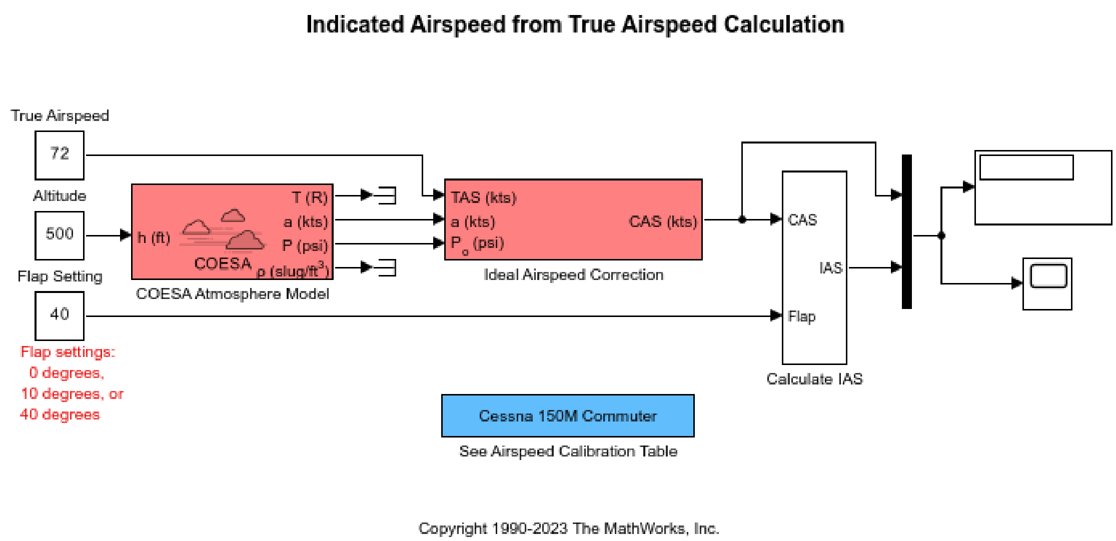
See Also

3x3 Cross Product | Quaternion Normalize | MATLAB Function

Indicated Airspeed from True Airspeed Calculation

This model shows how to compute the indicated airspeed (IAS) from true airspeed using the Ideal Airspeed Correction block. The Aerospace Blockset™ blocks are indicated in red.

```
open_system('aeroblk_indicated');
snapshotModel('aeroblk_indicated');
```



Provide a True Airspeed

True airspeed is the airspeed that we would read ideally. You can set the true airspeed in the True Airspeed block in the model.

Calculate the Calibrated Airspeed

To calculate the calibrated airspeed, you adjust the true airspeed for errors introduced through the pitot-static airspeed indicators used to determine airspeed. These measurement errors are density error, compressibility error, and calibration error. The Ideal Airspeed Correction block can apply the density error and compressibility error computing calibrated airspeed from true airspeed.

Density Error

An airspeed indicator reads lower than true airspeed at higher altitudes. This is due to lower air density at altitude. When the difference or error in air density at altitude from air density on a standard day at sea level is applied to true airspeed, the result is in equivalent airspeed (EAS). Equivalent airspeed is true airspeed modified with the changes in atmospheric density that affect the airspeed indicator.

Compressibility Error

Air has a limited ability to resist compression. This ability is reduced by an increase in altitude, an increase in speed, or a restricted volume. Within the airspeed indicator, there is a certain amount of trapped air. When flying at high altitudes and higher airspeeds, calibrated airspeed (CAS) is always higher than equivalent airspeed. Calibrated airspeed is equivalent airspeed modified with compressibility effects of air, which affect the airspeed indicator. True airspeed is the airspeed that we would read ideally (and the airspeed value easily calculated within a simulation). However there are errors introduced through the pitot-static airspeed indicators used to determine airspeed. These measurement errors are density error, compressibility error and calibration error. Applying these errors to true airspeed will result in indicated airspeed (the ideal airspeed correction block can handle the density error and compressibility error).

Adjust to Indicated Airspeed for Pitot-Static Airspeed Indicator

Calibration error of the pitot-static airspeed indicator is the last adjustment to the airspeed value. The adjustment results in an indicated airspeed displayed on the airspeed indicator in the cockpit.

Calibration Error

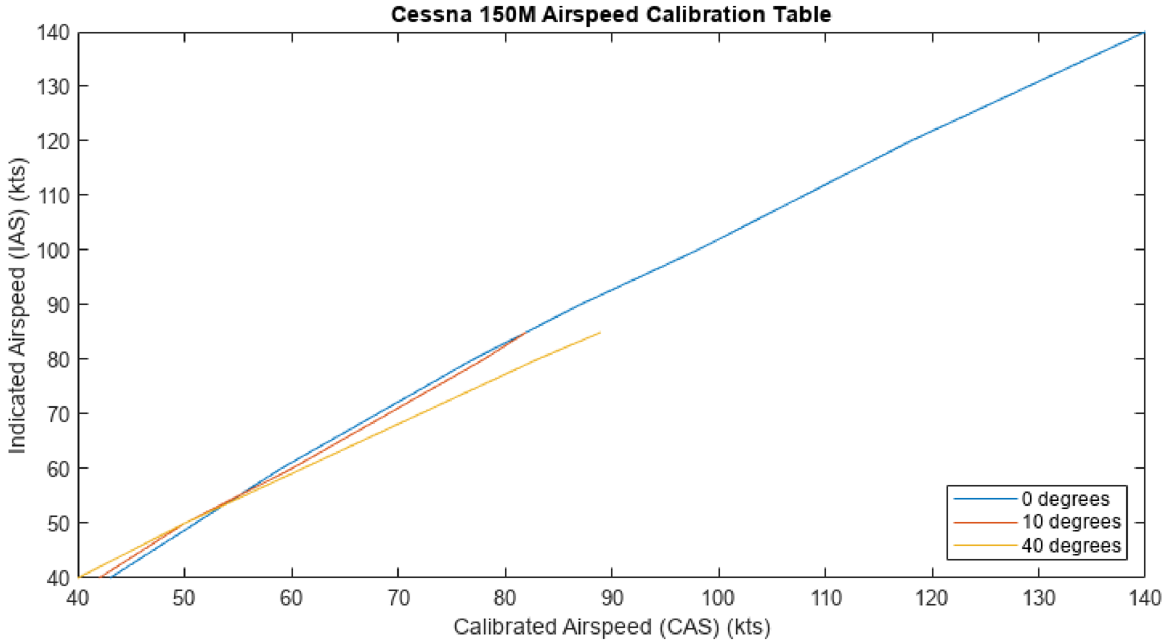
The airspeed indicator has static vent(s) to maintain a pressure equal to atmospheric pressure inside the instrument. Position and placement of the static vent, angle of attack, and velocity of the aircraft determines the pressure inside the airspeed indicator and the amount of calibration error of the airspeed indicator. The calibration error is specific to a given aircraft design. A calibration table is usually given in the pilot operating handbook (POH) or in other aircraft specifications.

Example Calibration Tables

As an example, here is the Cessna 150M airspeed calibration table from "Pilot's Operating Handbook, Cessna 1976 150 Commuter, Cessna Model 150M", Cessna Aircraft Company, Wichita, Kansas, USA, 1976.

Calibration table for 0, 10, and 0 degrees of flap:

```
flaps0IAS = 40:10:140;
flaps0CAS = [43 51 59 68 77 87 98 108 118 129 140];

flaps10IAS = [40:10:80 85];
flaps10CAS = [42 50 60 69 78 82];

flaps40IAS = [40:10:80 85];
flaps40CAS = [40 50 61 72 83 89];

plot(flaps0CAS,flaps0IAS,flaps10CAS,flaps10IAS,flaps40CAS,flaps40IAS)
xlabel('Calibrated Airspeed (CAS) (kts)');
ylabel('Indicated Airspeed (IAS) (kts)');
title('Cessna 150M Airspeed Calibration Table');
legend ('0 degrees','10 degrees','40 degrees','Location','southeast');
```



Simulate Model to Display Airspeeds

You can see this indicated airspeed adjustment based on these example calibration tables modeled in the Calculate IAS block. After simulating the model, indicated airspeed is displayed with calibrated airspeed in the scope and the display block.

See Also

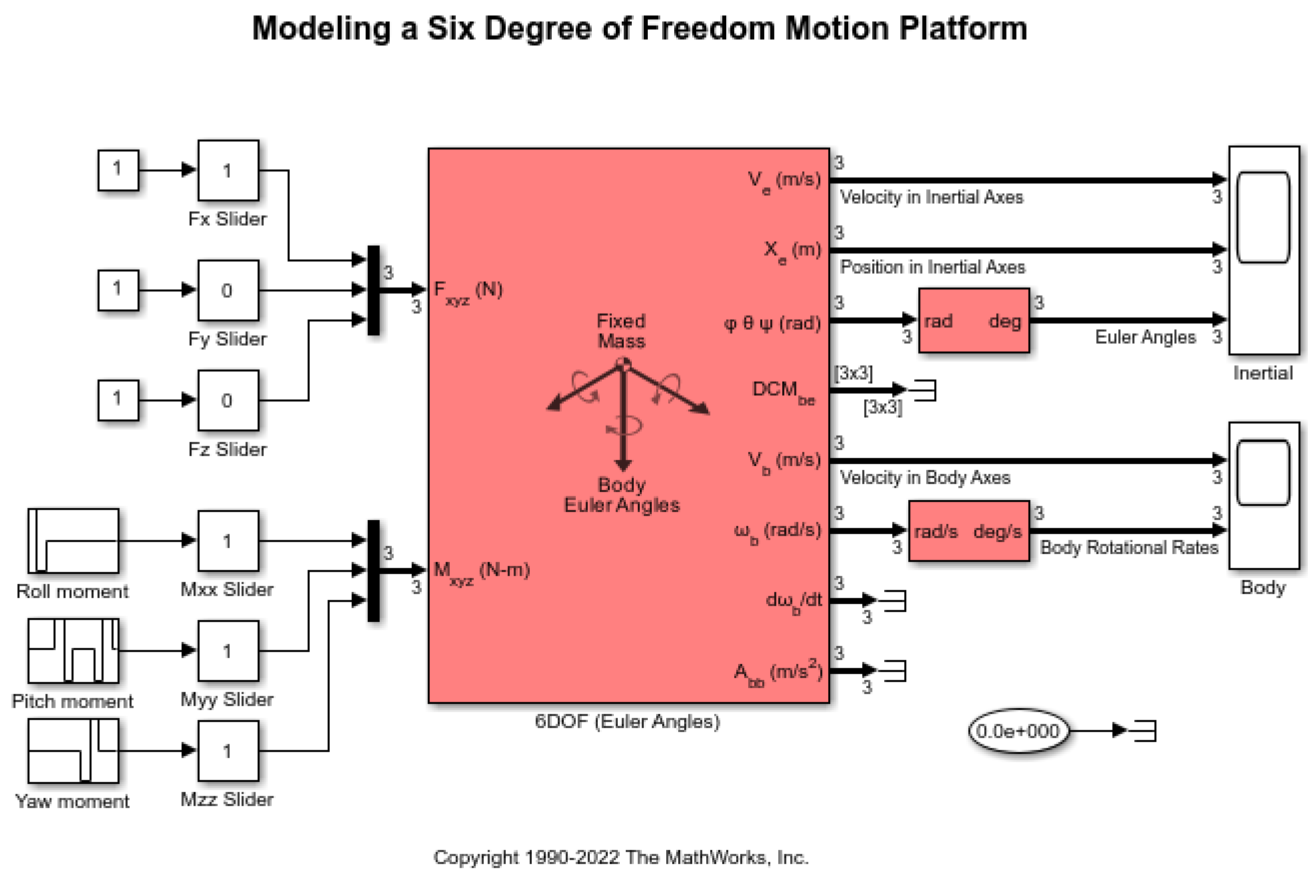
Ideal Airspeed Correction | COESA Atmosphere Model

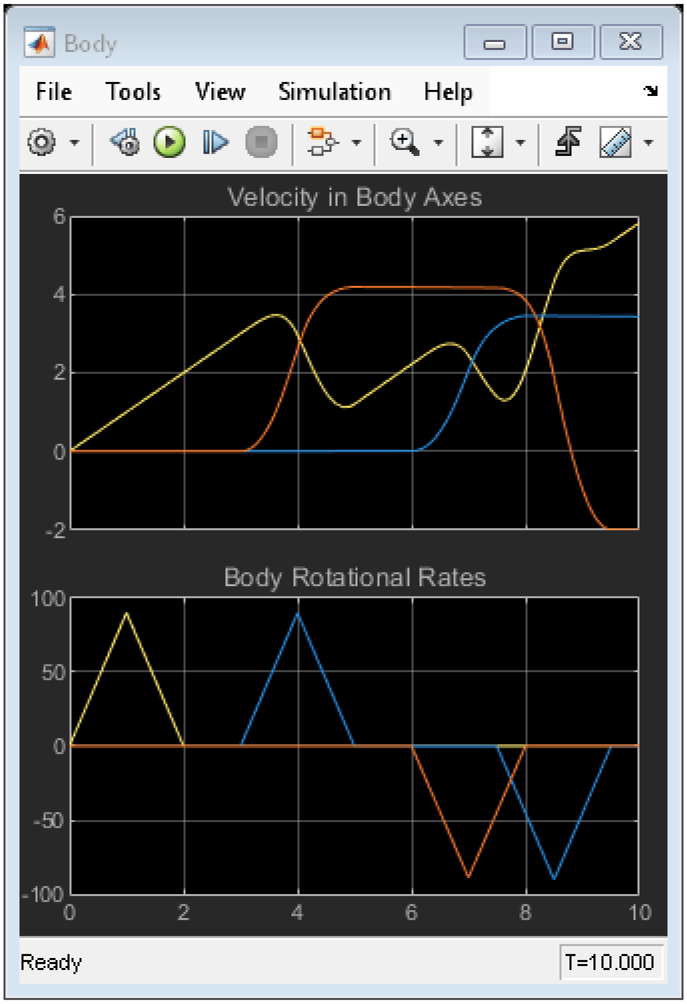
Related Examples

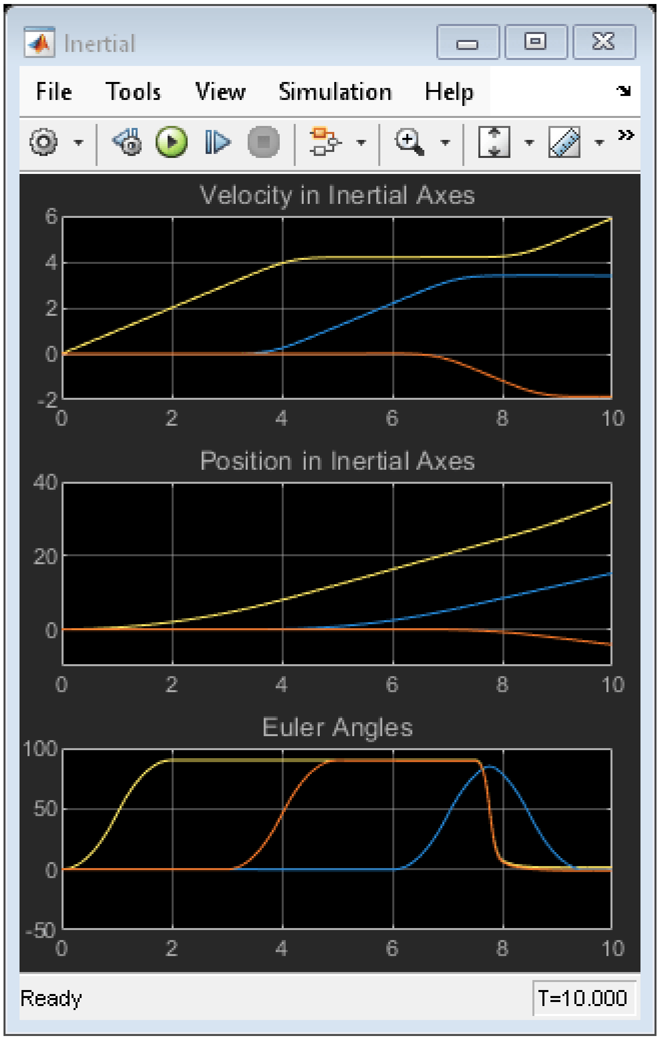
- “Ideal Airspeed Correction” on page 3-2

Modeling a Six Degree of Freedom Motion Platform

This example shows how to connect an Aerospace Blockset™ six degree of freedom equation of motion block.




Body
File Tools View Simulation Help
Velocity in Body Axes
6
4
2
0
-2
Body Rotational Rates
100
50
0
-50
-100
0 2 4 6 8 10
Ready
T=10.000



See Also

6DOF (Euler Angles)

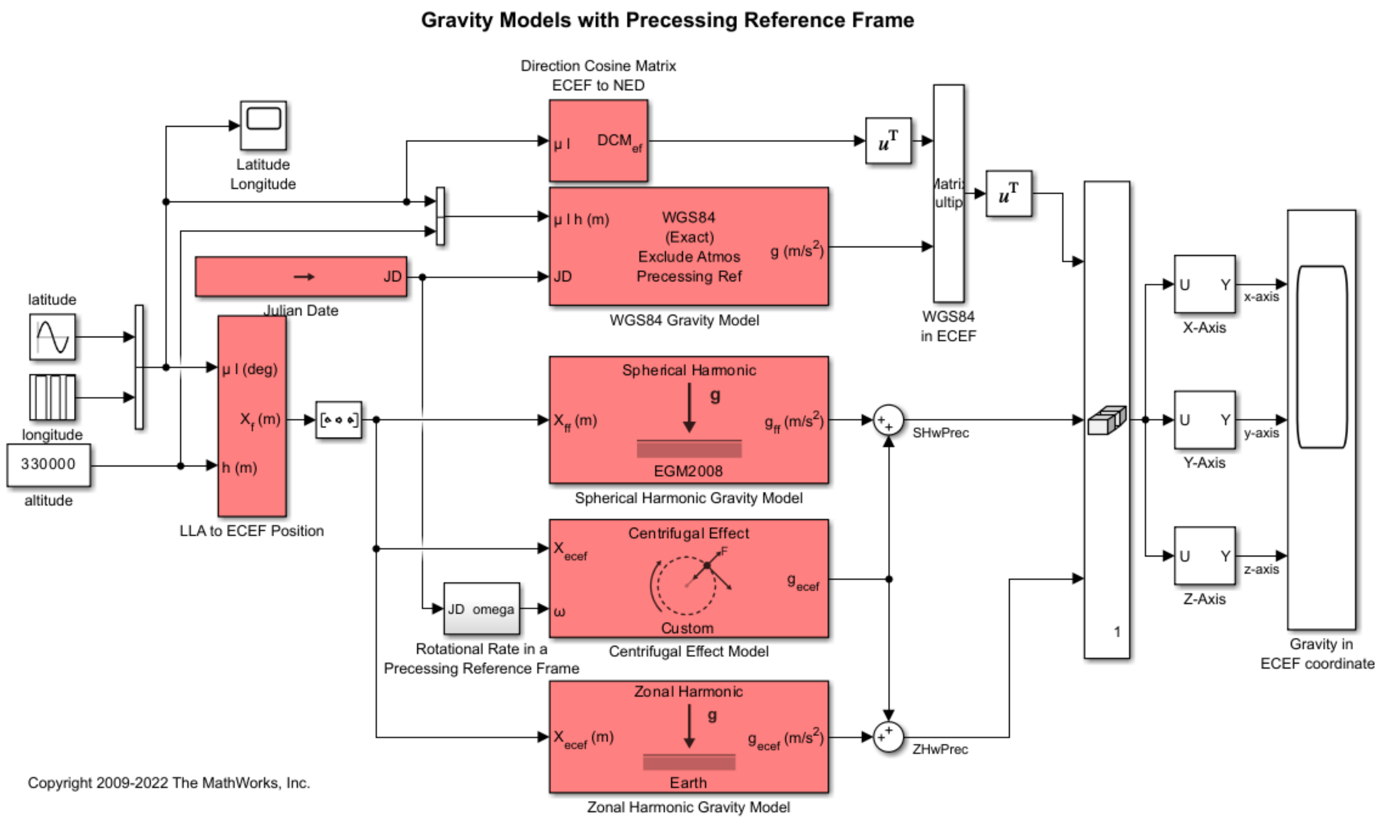
Gravity Models with Precessing Reference Frame

This example shows how to implement various gravity models with precessing reference frames using Aerospace Blockset™ blocks. Precession is a change in the orientation of the rotational axis of a rotating body. The orientation of the rotational axis of Earth determines the angular velocity experienced at different positions, which in turn affects the computed centrifugal forces. The Aerospace Blockset blocks are shown in red.

Open the Example Model

The example `asbGravWPrec.slx` is configured to simulate a reference trajectory that is 330 km over the Earth with a ground track along the prime meridian and a period of 10 seconds.

```
open_system('asbGravWPrec.slx');
```



Summary

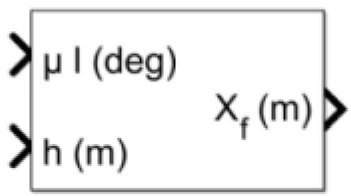
This example shows how to compute gravity forces using these gravity model blocks: WGS84, spherical harmonic, and zonal harmonic gravity model.

- Since spherical and zonal harmonic models require positions in the Earth-Centered Earth-Fixed (ECEF) frame, the example needs to perform a coordinate change from geodetic latitude and longitude to ECEF coordinates.
- Zonal and spherical harmonic gravity models also do not include centrifugal effects, so centrifugal forces needs to be computed separately and summed with the respective gravity models.

- To account for Earth precession and the influence it has on rotation rate, the centrifugal effect model block uses a specific rotation rate from a given Julian date to compute the centrifugal forces. The inclusion of the effects Julian date has on centrifugal forces is useful in long simulations where the effects of Earth's precession cannot be neglected.
- This example does not take into account coriolis acceleration.
- The WGS84 gravity model outputs gravity components in the North-East Down (NED) frame. To compare gravity models, the example computes a Direction Cosine Matrix (DCM) to convert the results into the ECEF frame to be consistent with the computed gravity values from the other two models.

Determine Earth-Centered Earth-Fixed (ECEF) Position

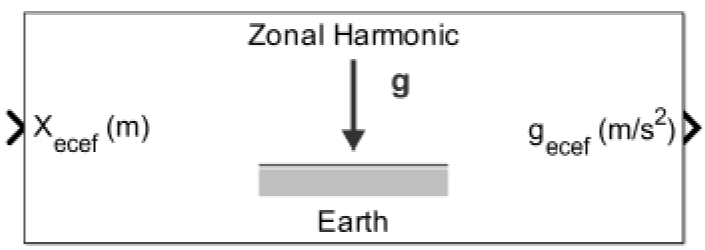
To use the zonal harmonic and spherical harmonic gravity models, you need positions in the ECEF reference frame. Use the LLA to ECEF Position block to determine ECEF positions from geodetic latitude, longitude, and altitude.



LLA to ECEF Position

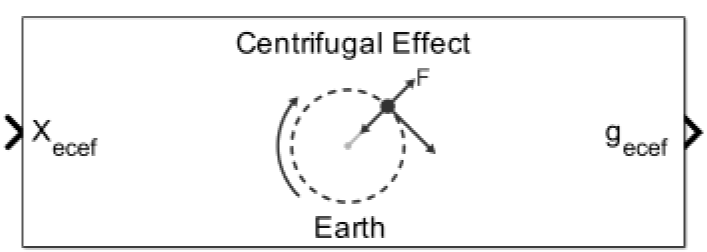
Compute Zonal Harmonic Gravity for Earth

Use the Zonal Harmonic Gravity Model block to calculate the gravity components up to the fourth order in the ECEF coordinate system in meters per second squared.



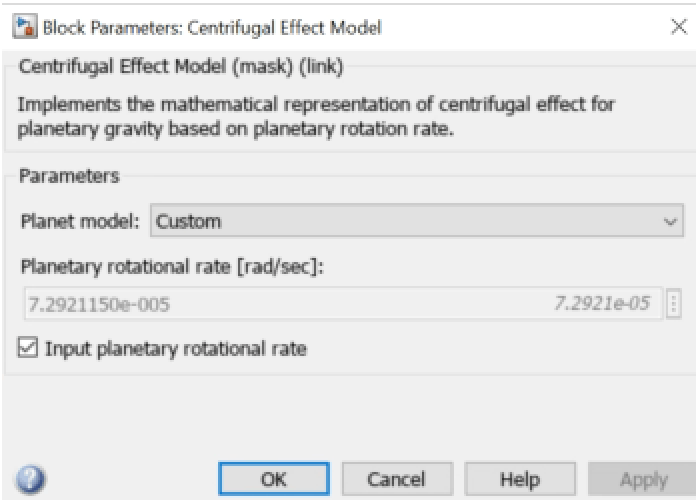
Zonal Harmonic Gravity Model

Since the block does not include centrifugal effects of planetary rotation and the effects of a precessing reference frame, use the Centrifugal Effect Model Block to compute the gravity components due to centrifugal effects.



Centrifugal Effect Model

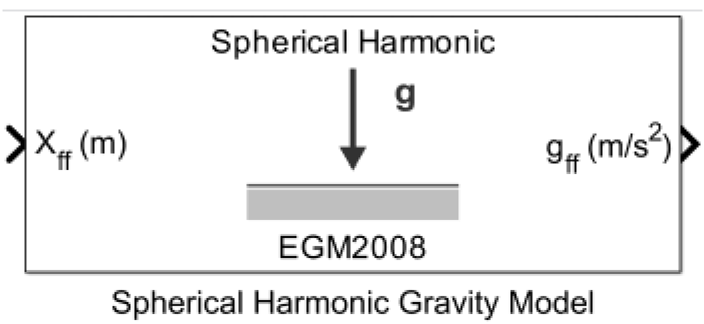
To include the effects of precession, select the **Input planetary rotational rate** check box.



Selecting this check box creates an additional input port to the block to specify a rotational rate. The changes that the Earth precession has on its rotational rate can be accurately approximated as a function of Julian date through the equations in the subsystem **Rotational Rate in a Precessing Reference Frame**.

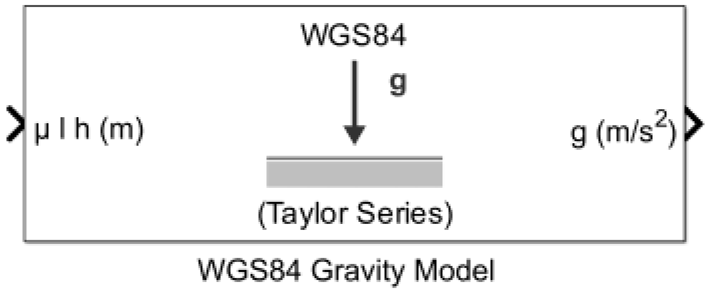
Compute Spherical Harmonic Gravity for Earth

To compute the gravity components in the ECEF coordinate system in meters per second squared, use the Spherical Harmonic Gravity Model block with the default EGM2008 central body model selected and a default degree of 120. This block does not include centrifugal effects of planetary rotation. To get total gravity potentials, add the gravity components of the centrifugal effects to the computed values.

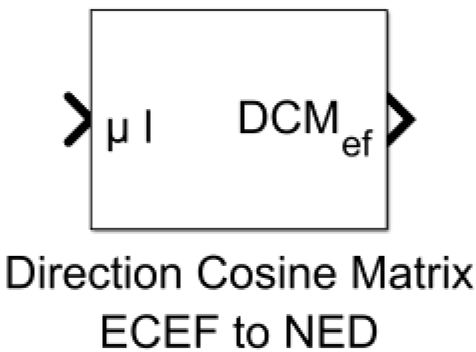


Calculate WGS84 Gravity for Earth

Use the WGS84 Gravity Model to calculate zonal harmonic gravity components in the NED coordinate system in meters per second squared. By default, the WGS84 Exact gravity model does not include centrifugal effects. To include centrifugal effects, clear the **No centrifugal effects** check box.



To compare the gravity models, their outputs must be in the same coordinate system. You can transform WGS84 gravity from NED coordinates to ECEF Coordinates by multiplying NED coordinates with a direction cosine matrix (DCM). This matrix is a 3-by-3 transformation matrix that converts values between two coordinate frames. Use a Direction Cosine Matrix ECEF to NED block to generate the DCM. This block accepts an input latitude and longitude and returns the DCM required to convert coordinates in the ECEF coordinate frame to the NED coordinate frame.



Since the resultant DCM is for a conversion from the ECEF to NED coordinate frame, take the transpose of the resultant matrix using the transpose block to generate the NED to ECEF DCM.

Comparison Plots for Different Gravity Models with Centrifugal Effects

Simulate the model and observe the latitude (in yellow) and longitude (in blue) position history of the trajectory.

```
sim('asbGravWPrec.slx');
```

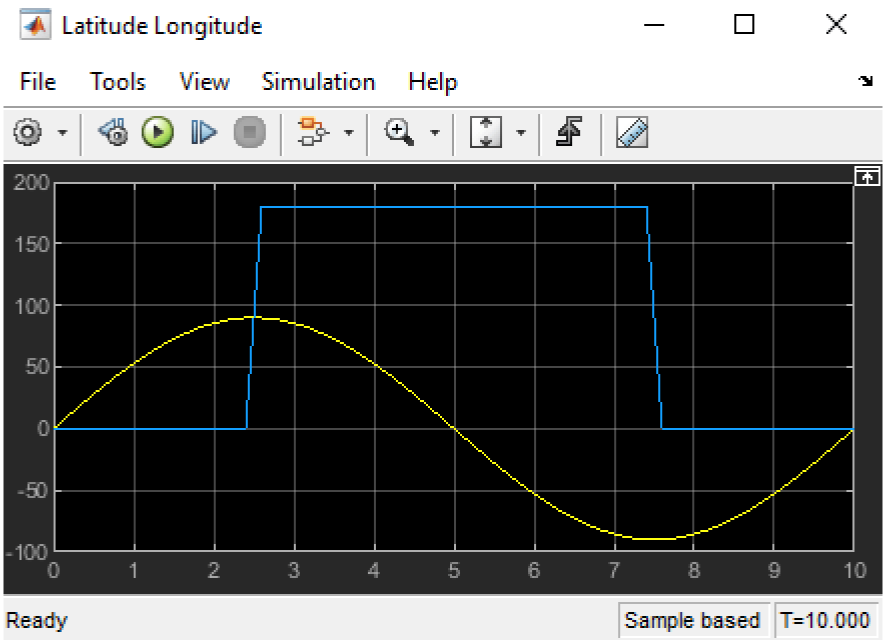


Figure 1: Latitude and longitude position history

This figure shows the time history gravity components each gravity model computes. Observe that when computing gravity at various points along a trajectory, the results between the models are similar. While the differences are relatively small, it is important to note that the simulation trajectory is in an open loop system (the trajectory is not computed using gravity), and if considered in a closed loop system, the impact will become more significant over time.

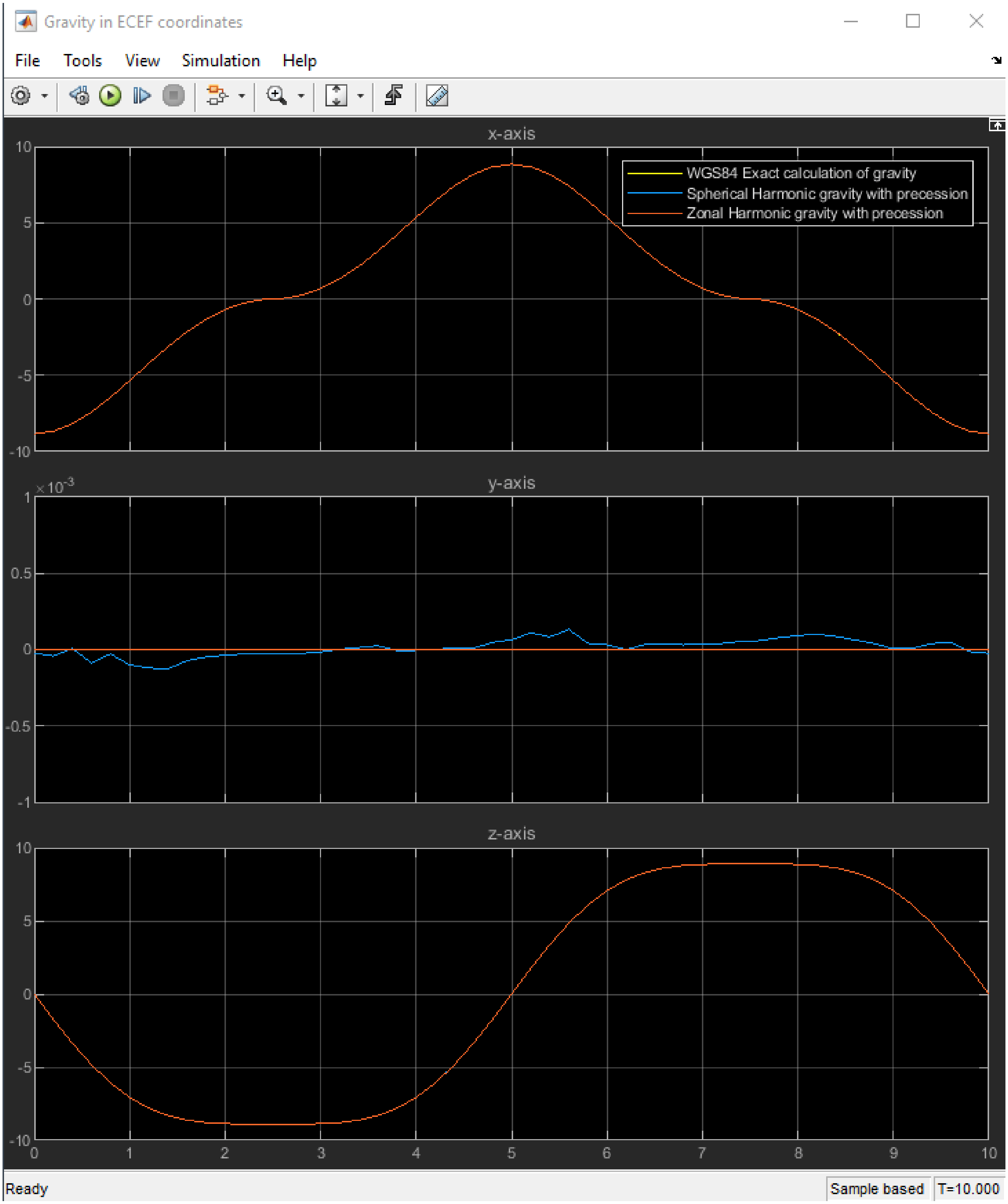

Gravity in ECEF coordinates
File
Tools
View
Simulation
Help
x-axis
WGS84 Exact calculation of gravity
Spherical Harmonic gravity with precession
Zonal Harmonic gravity with precession
y-axis
$\times 10^{-3}$
z-axis
Ready
Sample based
T=10.000

Figure 2: Gravity components in the Earth-Centered Earth-Fixed coordinate frame

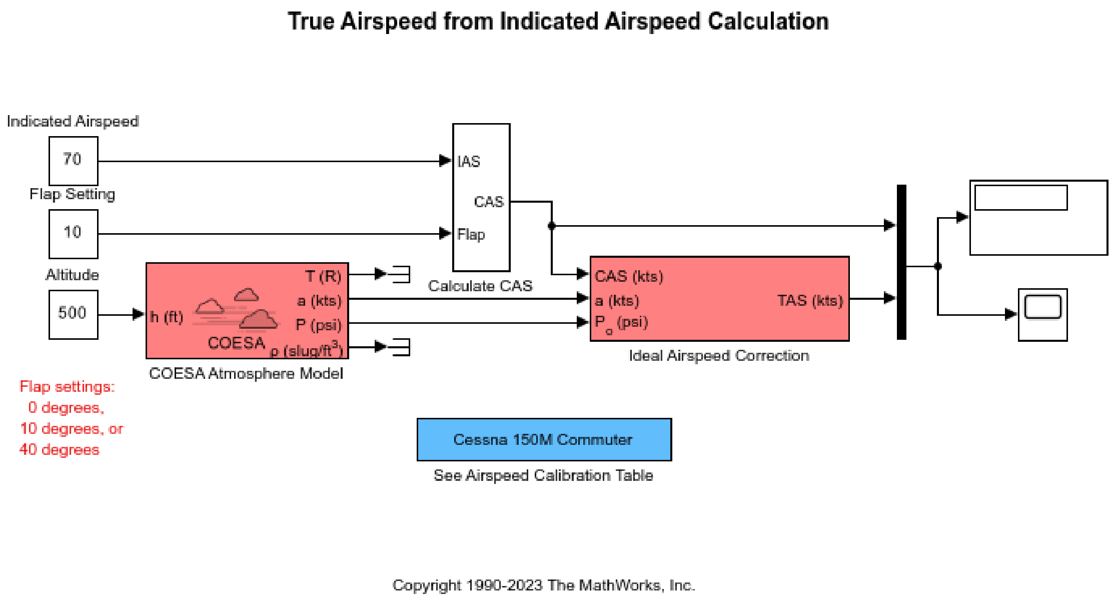
See Also

WGS84 Gravity Model | Spherical Harmonic Gravity Model | Centrifugal Effect Model | Zonal Harmonic Gravity Model

True Airspeed from Indicated Airspeed Calculation

This model shows how to compute true airspeed from indicated airspeed using the Ideal Airspeed Correction block. The Aerospace Blockset™ blocks are indicated in red.

```
open_system('aeroblk_calibrated');
snapshotModel('aeroblk_calibrated');
```



Calculating True Airspeed

True airspeed (TAS) is the airspeed that we would read ideally (and the airspeed value easily calculated within a simulation). However there are errors introduced through the pitot-static airspeed indicators used to determine airspeed. These measurement errors are density error, compressibility error and calibration error. Removing these errors from indicated airspeed (IAS) will result in true airspeed.

Correcting Indicated Airspeed for Calibration Error

An airspeed indicator has static vent(s) to maintain a pressure equal to atmospheric pressure inside the instrument. Position and placement of the static vent along with angle of attack and velocity of the aircraft will determine the pressure inside the airspeed indicator and thus the amount of calibration error of the airspeed indicator. Needless to say, calibration error is specific to a given aircraft design. A calibration table is usually given in the pilot operating handbook (POH) or in other aircraft specifications. Using this calibration table, the calibrated airspeed (CAS) is determined from indicated airspeed by removing the calibration error of the airspeed indicator. Indicated airspeed is displayed in the cockpit instrumentation.

This example is using the airspeed calibration table for the Cessna 150M from "Pilot's Operating Handbook, Cessna 1976 150 Commuter, Cessna Model 150M", Cessna Aircraft Company, Wichita, Kansas, USA, 1976.

Correcting Calibrated Airspeed for Compressibility and Density Errors

The Ideal Airspeed Correction block is used to correct calibrated airspeed to true airspeed by removing the compressibility error and the density error.

Compressibility Error

Air has a limited ability to resist compression. This ability is reduced by an increase in altitude, an increase in speed, or a restricted volume. Within the airspeed indicator, there is a certain amount of trapped air. When flying at high altitudes and higher airspeeds, calibrated airspeed is always higher than equivalent airspeed (EAS). Equivalent airspeed is calibrated airspeed with compressibility effects of air which affect the airspeed indicator removed.

Density Error

An airspeed indicator reads lower than true airspeed at higher altitudes. This is due to lower air density at altitude. When the difference or error in air density at altitude from air density on a standard day at sea level is applied to true airspeed, it results in equivalent airspeed. True airspeed is equivalent airspeed with the changes in atmospheric density which affect the airspeed indicator removed.

Simulate Model to Display Airspeeds

You can see the true airspeed based on the example calibration tables modeled in the Calculate CAS block. After simulating the model, true airspeed is displayed with calibrated airspeed in the scope and the display block.

See Also

Ideal Airspeed Correction | COESA Atmosphere Model

Related Examples

- “Ideal Airspeed Correction” on page 3-2

Airframe Trim and Linearize with Simulink Control Design

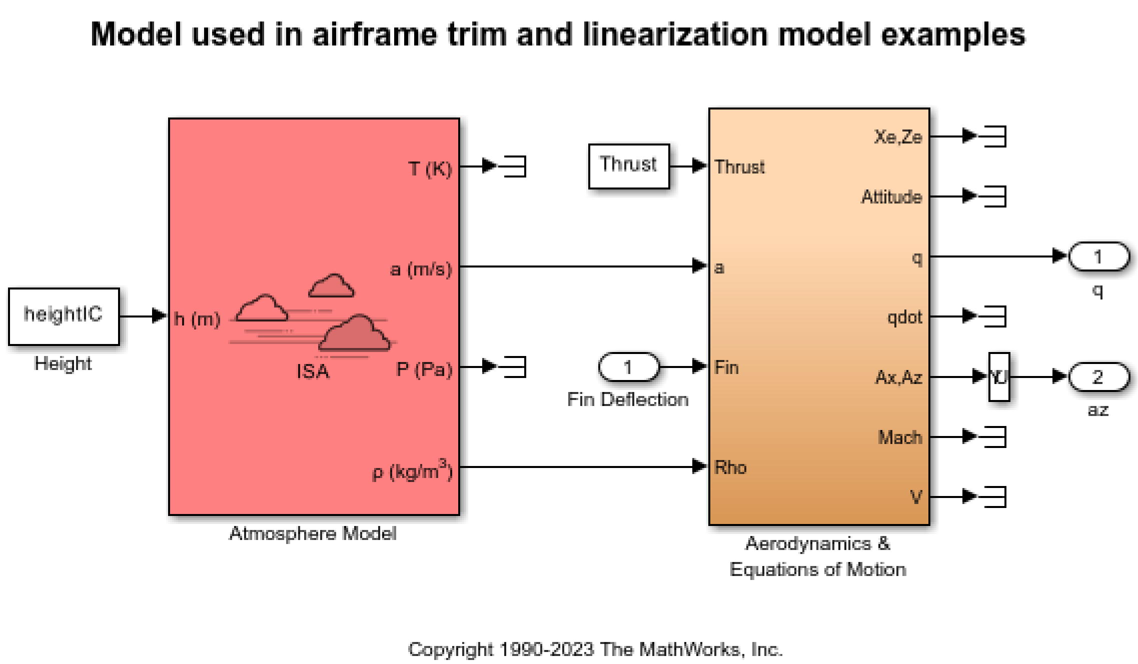
This example shows how to trim and linearize an airframe using Simulink® Control Design™ software.

Designing an autopilot with classical design techniques requires linear models of the airframe pitch dynamics for several trimmed flight conditions. The MATLAB® technical computing environment can determine the trim conditions and derive linear state-space models directly from the nonlinear Simulink® and Aerospace Blockset™ model. This step saves time and helps to validate the model. The Simulink Control Design functions allow you to visualize the motion of the airframe in terms of open-loop frequency or time responses.

Initialize Guidance Model

The first problem is to find the elevator deflection, and the resulting trimmed body rate (q), which will generate a given incidence value when the missile is traveling at a set speed. Once the trim condition is found, a linear model can be derived for the dynamics of the perturbations in the states around the trim condition.

```
open_system('aeroblk_guidance_airframe');
```



Define State Values

Define the state values for trimming:

- Height [m]
- Mach Number
- Incidence [rad]

- Flightpath Angle [rad]
- Total Velocity [m/s]
- Initial Pitch Body Rate [rad/sec]

```
heightIC    = 10000/m2ft;
machIC      = 3;
alphaIC     = -10*d2r;
thetaIC     = 0*d2r;
velocityIC  = machIC*(340+(295-340)*heightIC/11000);
pitchRateIC = 0;
```

Set Operating Point and State Specifications

The first state specifications are Position states. The second state specification is Theta. Both are known, but not at steady state. The third state specifications are body axis angular rates, of which the variable w is at steady state.

```
opspec = operspec('aeroblk_guidance_airframe');
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
```

Search for Operating Point, Set I/O, Then Linearize

Find the operating point that meets the state specification and then derive the linear model using the specified input and outputs.

```
op = findop('aeroblk_guidance_airframe',opspec);

io(1) = linio('aeroblk_guidance_airframe/Fin Deflection',1,'input');
io(2) = linio('aeroblk_guidance_airframe/Selector',1,'output');
io(3) = linio(sprintf(['aeroblk_guidance_airframe/Aerodynamics &\n', ...
                       'Equations of Motion']),3,'output');

sys = linearize('aeroblk_guidance_airframe',op,io);
```

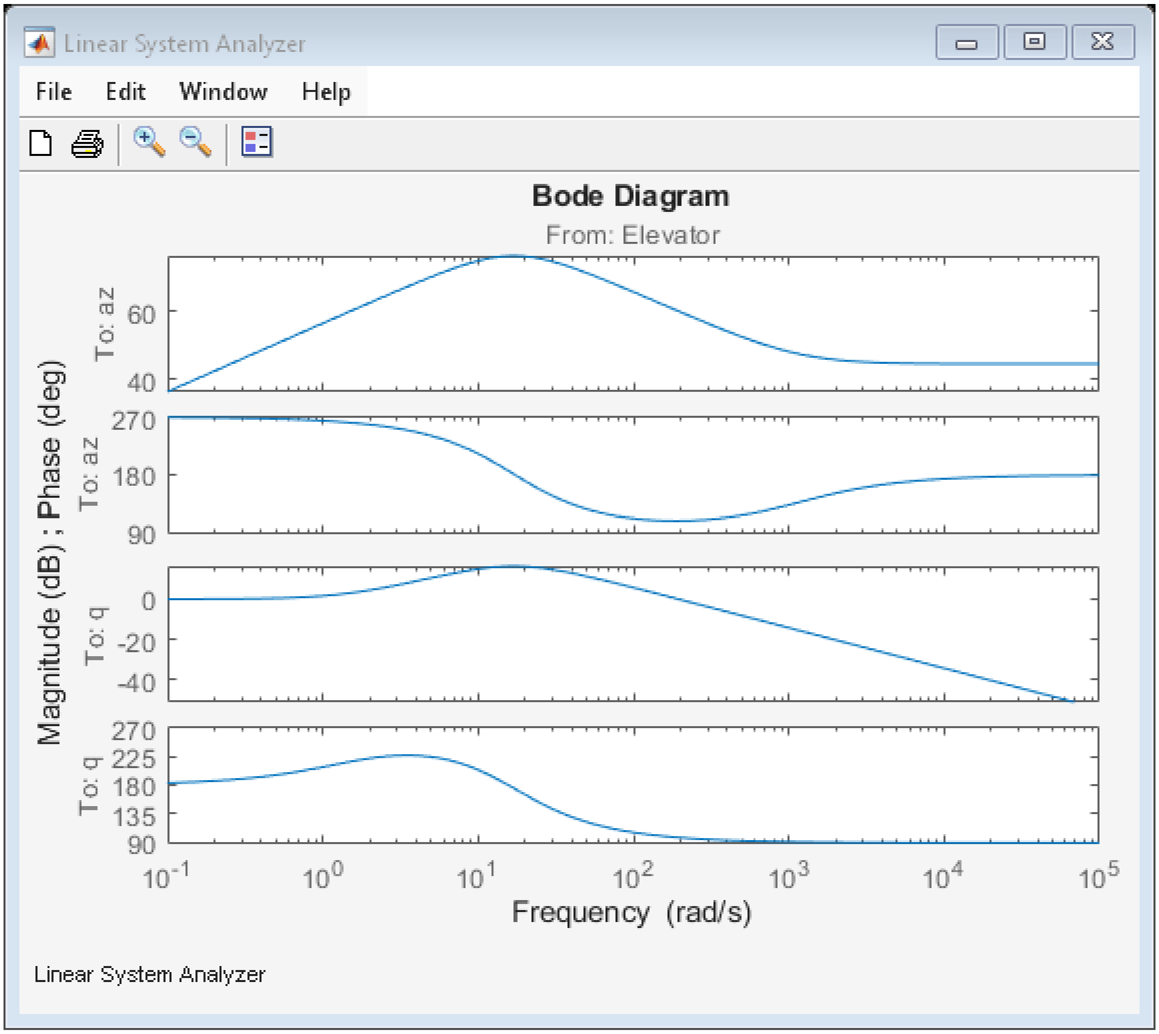
```
 Operating point search report:
---------------------------------

opreport =


 Operating point search report for the Model aeroblk_guidance_airframe.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
   Min          x          Max        dxMin         dx        dxMax
__________ __________ __________ __________ __________ __________

(1.) aeroblk_guidance_airframe/Aerodynamics & Equations of Motion/3DOF (Body Axes)/Position
    0          0          0         -Inf     967.6649     Inf
```

```
-3047.9999 -3047.9999 -3047.9999    -Inf    -170.6254     Inf
(2.) aeroblk_guidance_airframe/Aerodynamics & Equations of Motion/3DOF (Body Axes)/Theta
    0          0          0         -Inf     -0.21604     Inf
(3.) aeroblk_guidance_airframe/Aerodynamics & Equations of Motion/3DOF (Body Axes)/U,w
 967.6649   967.6649   967.6649     -Inf     -14.0977     Inf
-170.6254  -170.6254  -170.6254      0      -7.439e-08     0
(4.) aeroblk_guidance_airframe/Aerodynamics & Equations of Motion/3DOF (Body Axes)/q
   -Inf     -0.21604     Inf         0      3.3582e-08     0

Inputs:
----------
  Min      u      Max
_______ _______ _______

(1.) aeroblk_guidance_airframe/Fin Deflection
 -Inf   0.13615   Inf

Outputs:
----------
   Min         y         Max
__________ __________ __________

(1.) aeroblk_guidance_airframe/q
   -Inf     -0.21604     Inf
(2.) aeroblk_guidance_airframe/az
   -Inf    -7.439e-08    Inf
```

Select Trimmed States, Create LTI Object, and Plot Bode Response

Find index of desired states in the state vector. Then create the LTI object and view the frequency response as a Bode diagram.

```
names = sys.StateName;
q_idx = find(strcmp('q',names));
az_idx = find(strcmp('U,w(2)',names));

airframe = ss(sys.A([az_idx q_idx],[az_idx q_idx]),sys.B([az_idx q_idx],:),sys.C(:,[az_idx q_idx

set(airframe,'inputname',{'Elevator'}, ...
             'outputname',[{'az'} {'q'}]);

linearSystemAnalyzer('bode',airframe);
```



Copyright 1990-2023 The MathWorks, Inc.

See Also

Blocks
ISA Atmosphere Model

Functions
`operspec` | `linio` | `linearize`

Airframe Trim and Linearize with Control System Toolbox

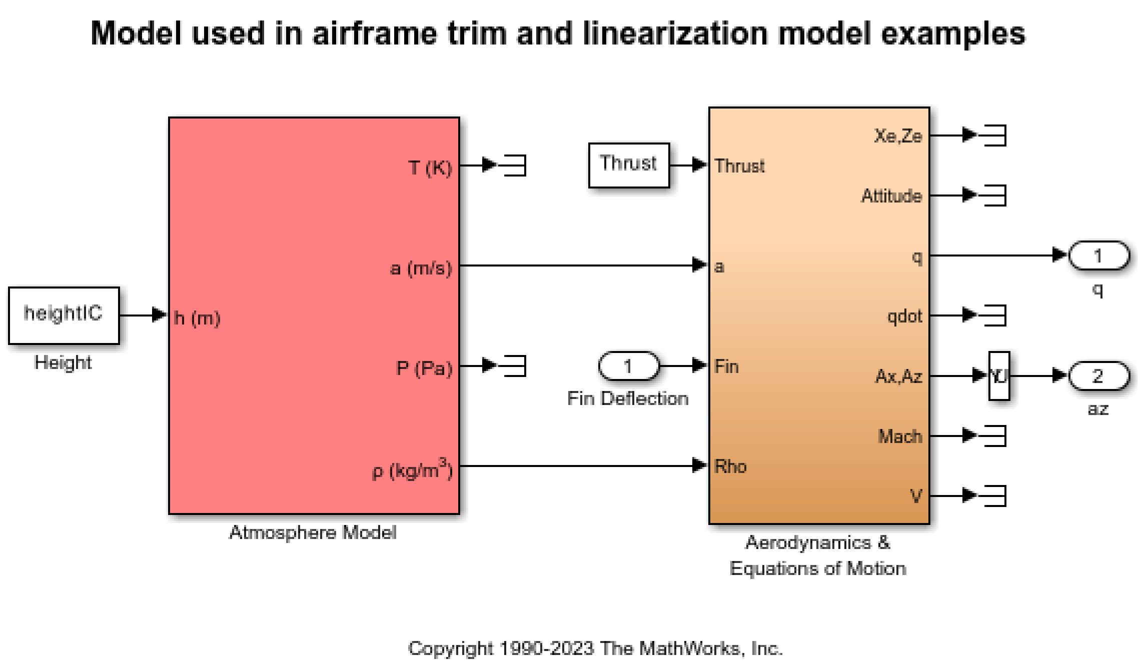
This example shows how to trim and linearize an airframe in the Simulink® environment using Control System Toolbox™.

Designing an autopilot with classical design techniques requires linear models of the airframe pitch dynamics for several trimmed flight conditions. The MATLAB® technical computing environment can determine the trim conditions and derive linear state-space models directly from the nonlinear model. This step saves time and helps to validate the model. The Control System Toolbox functions allow you to visualize the motion of the airframe in terms of open-loop frequency or time responses.

Initialize Guidance Model

Find the elevator deflection and the resulting trimmed body rate (q). These calculations generate a given incidence value when the airframe is traveling at a set speed. Once the trim condition is found, a linear model can be derived for the dynamics of the perturbations in the states around the trim condition.

```
open_system('aeroblk_guidance_airframe');
```



Define State Values

Define the state values for trimming:

- Height [m]
- Mach Number
- Incidence [rad]
- Flightpath Angle [rad]

- Total Velocity [m/s]
- Initial Pitch Body Rate [rad/sec]

```
heightIC    = 10000/m2ft;
machIC      = 3;
alphaIC     = -10*d2r;
thetaIC     = 0*d2r;
velocityIC  = machIC*(340+(295-340)*heightIC/11000);
pitchRateIC = 0;
```

Find Names and Ordering of States

Find the names and the ordering of states from the model.

```
[sizes,x0,names]=aeroblk_guidance_airframe([],[],[],'sizes');

state_names = cell(1,numel(names));
for i = 1:numel(names)
    n = max(strfind(names{i},'/'));
    state_names{i}=names{i}(n+1:end);
end
```

Specify States

Specify which states to trim and which states remain fixed.

```
fixed_states            = [{'U,w'} {'Theta'} {'Position'}];
fixed_derivatives       = [{'U,w'} {'q'}];
fixed_outputs           = [];
fixed_inputs            = [];

n_states=[];n_deriv=[];
for i = 1:length(fixed_states)
    n_states=[n_states find(strcmp(fixed_states{i},state_names))]; %#ok<AGROW>
end
for i = 1:length(fixed_derivatives)
    n_deriv=[n_deriv find(strcmp(fixed_derivatives{i},state_names))]; %#ok<AGROW>
end
n_deriv=n_deriv(2:end);                             % Ignore U
```

Trim Model

Trim the model.

```
[X_trim,U_trim,Y_trim,DX]=trim('aeroblk_guidance_airframe',x0,0,[0 0 velocityIC]', ...
                               n_states,fixed_inputs,fixed_outputs, ...
                               [],n_deriv)  %#ok<NOPTS>
```

```
X_trim =

   1.0e+03 *

   -0.0002
         0
    0.9677
   -0.1706
         0
```

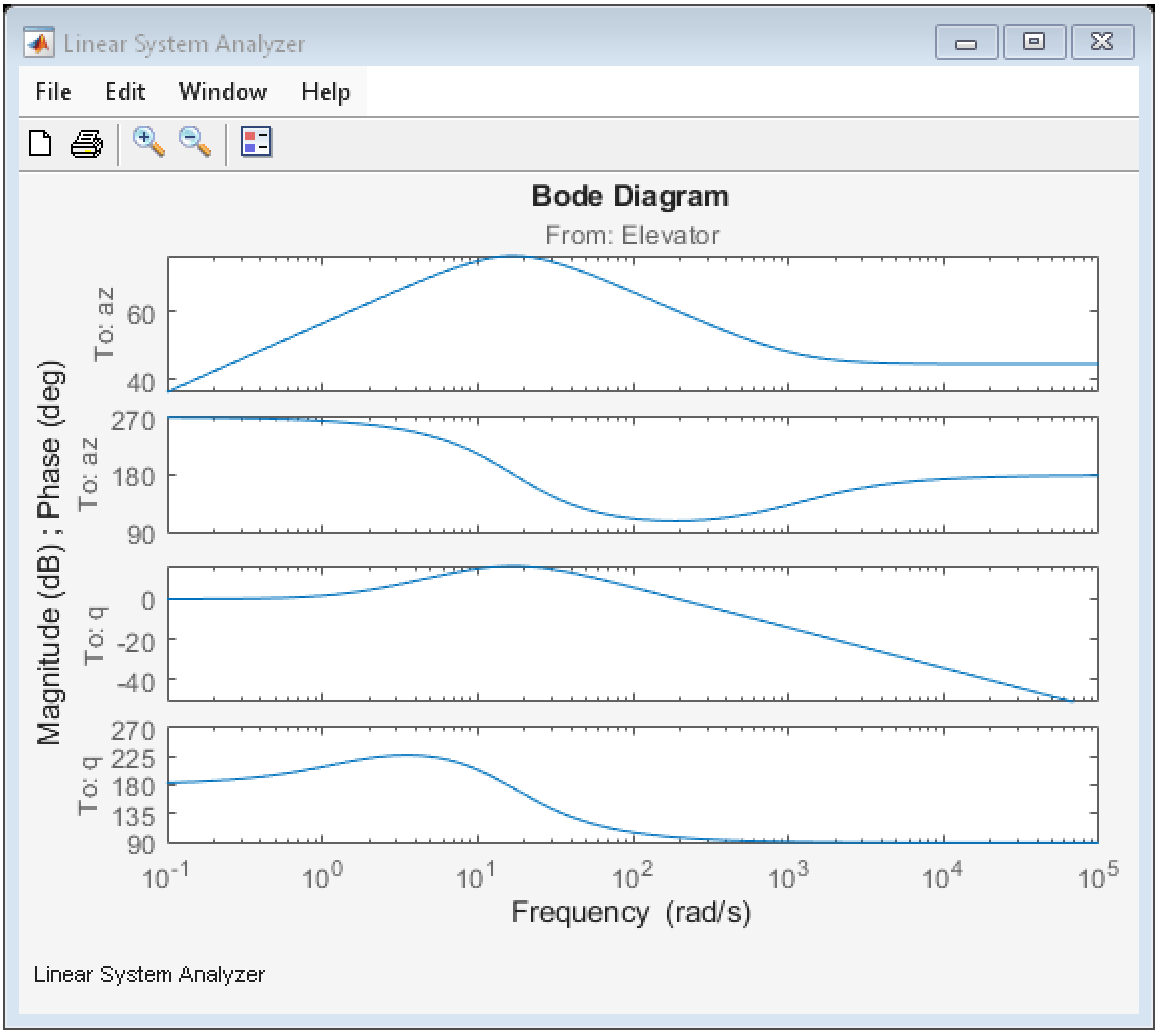
```
   -3.0480


U_trim =

    0.1362


Y_trim =

   -0.2160
         0


DX =

         0
   -0.2160
  -14.0977
         0
  967.6649
 -170.6254
```

Linear Model and Frequency Response

Derive the linear model and view the frequency response.

```
[A,B,C,D]=linmod('aeroblk_guidance_airframe',X_trim,U_trim);
if exist('control','dir')
    airframe = ss(A(n_deriv,n_deriv),B(n_deriv,:),C([2 1],n_deriv),D([2 1],:));
    set(airframe,'StateName',state_names(n_deriv), ...
        'InputName',{'Elevator'}, ...
        'OutputName',[{'az'} {'q'}]);

    zpk(airframe)
    linearSystemAnalyzer('bode',airframe)
end
```

```
ans =

  From input "Elevator" to output...
          -170.45 s (s+1133)
   az:  ----------------------
        (s^2 + 30.04s + 288.9)

         -194.66 (s+1.475)
   q:  ----------------------
       (s^2 + 30.04s + 288.9)

Continuous-time zero/pole/gain model.
```



See Also

Blocks
ISA Atmosphere Model

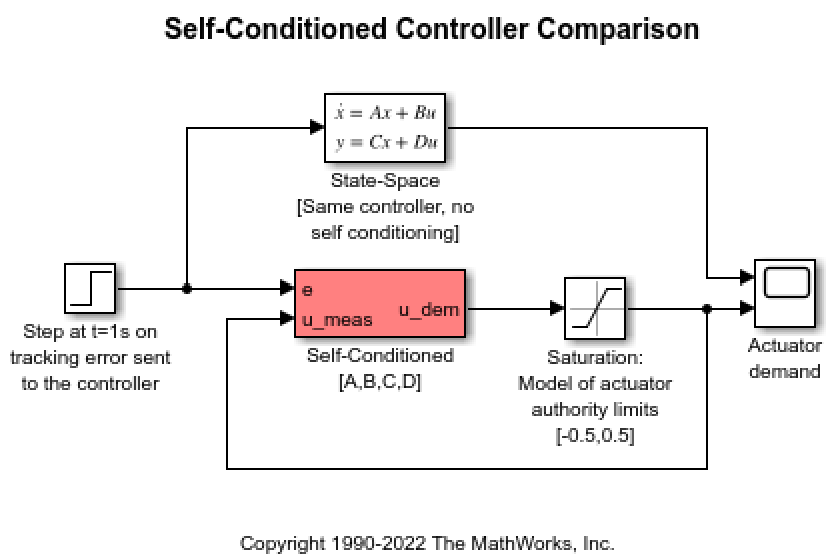
Functions
zpk

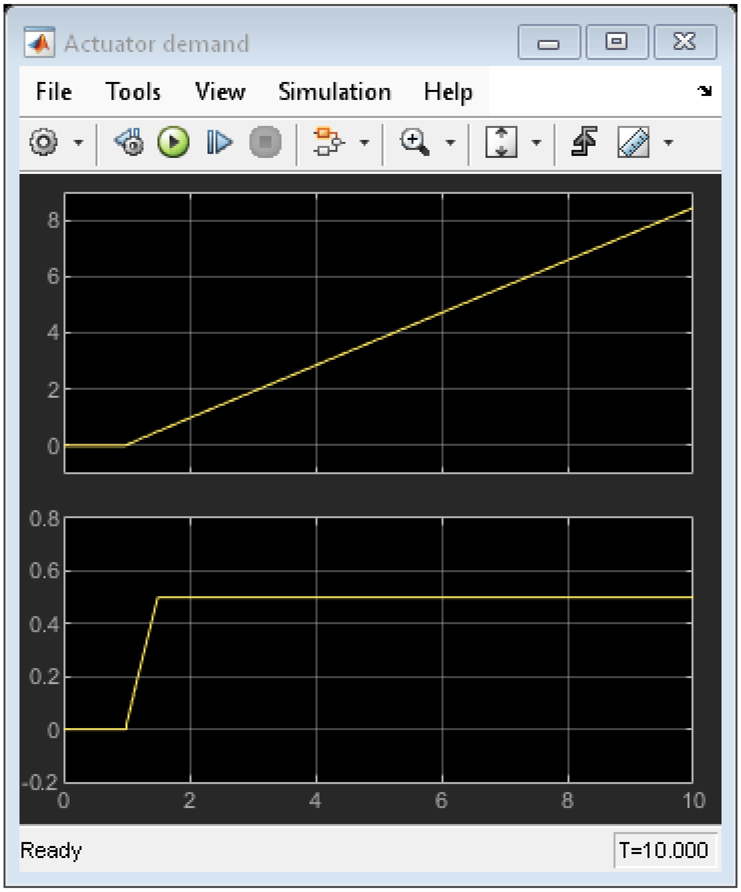
Self-Conditioned Controller Comparison

This example shows how to compare the implementation of a state-space controller [A,B,C,D] in a self-conditioned form versus a typical state-space controller [A,B,C,D]. This model requires Control System Toolbox™ software.

For the self-conditioned state-space controller, if measured control value is equal to the demanded control value (u_meas = u_dem), then the controller implementation is the typical state-space controller [A,B,C,D]. However if measured control value (u_meas) is limited, e.g., rate limiting, then the poles of the controller become those defined in the mask dialog box.

The results of a typical state-space controller [A,B,C,D] and a self-conditioned state-space controller with a limited measured control value are shown below.





See Also

Self-Conditioned [A,B,C,D]

Related Examples

- State-Space

Quadcopter Modeling and Simulation based on Parrot Minidrone

This example shows how to use Simulink® to model, simulate, and visualize a quadcopter, based on the Parrot® series of mini-drones.

For more details on the quadcopter implementation, see “Model a Quadcopter Based on Parrot Minidrones” on page 3-22.

Note: Hardware integration with the example would require installation of the Simulink Support Package for Parrot Minidrones and a C/C++ compiler.

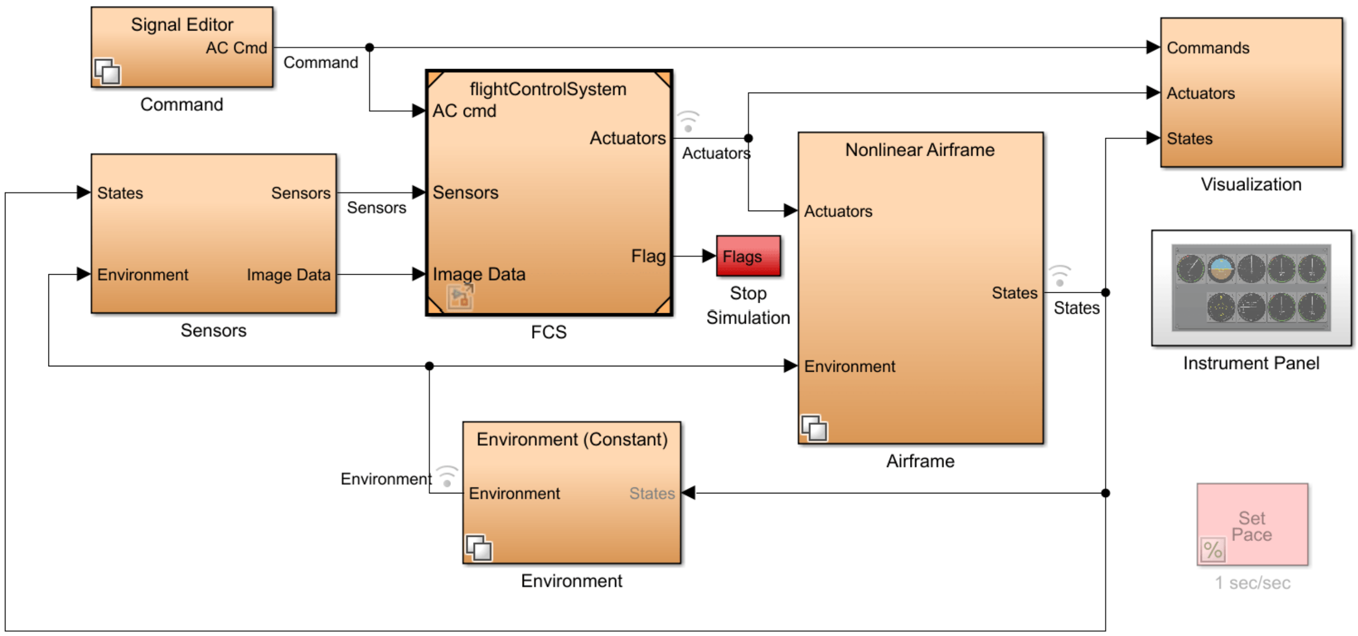
Open the Quadcopter Project

Run the following command to create and open a working copy of the project files for this example:

```
openProject('asbQuadcopter');
```

Design

The high-level design is as shown below, with each component designed as separate subsystems, and each subsystem supports multiple approaches, in the form of variants.



Command

Provides input command signals to all six degrees of freedom (x, y, z, roll, pitch, yaw). The VSS_COMMAND variable can be used to selectively provide the input command signal using:

- Signal Editor Block
- Joystick
- Pre-saved Data
- Spreadsheet

Environment

Sets the environmental parameters, gravity, temperature, speed of sound, pressure, density, and magnetic field to appropriate values. The VSS_ENVIRONMENT variable can be used to selectively set the environment parameter values to be:

- Constant
- Variable

Sensors

Measures the position, orientation, velocity, acceleration of the vehicle and also capture the visual data. The VSS_SENSORS variable can be used to selectively set the sensors to be:

- Dynamic
- Feedthrough

Airframe

Select the vehicle model, as either, nonlinear, or linear using the VSS_AIRFRAME variable. The linear state-space model is obtained by linearizing the non-linear model about a trim solution using the Simulink Control Design toolbox.

FCS (Flight Control System)

The FCS consists of:

- Estimators: implemented using a complementary filter and Kalman filter to estimate the position and orientation of the vehicle. The estimator parameters are based on [1].
- Flight Controller: implemented using PID controllers to control the position and orientation of the vehicle.
- Landing Logic: implements the algorithm to override the reference commands and land the vehicle in specific scenarios.

Visualization

The actuator input values, the vehicle states, and the environment parameter values are logged and can be visualized using the Simulation Data Inspector. The vehicle orientation and actuator input values are displayed using Flight Instruments.

In addition, the VSS_VISUALIZATION variable can be used to visualize the vehicle orientation using:

- Scopes
- In Workspace
- FlightGear
- Airport scene
- AppleHill scene: requires Unreal from Epic Games® and the Aerospace Blockset Interface for Unreal Engine® support package installed.

Implementation

- The `AC` cmd bus signal from the **Command** subsystem forms the reference signal to the **FCS** subsystem.
- The `Environment` bus signal from the **Environment** subsystem has the environment data and is passed on to the **Sensors** subsystem (acceleration due to gravity input for the Inertial Measurement Unit), and the **Airframe** subsystem (used in computation of forces and moments).
- The `Sensors` bus signal and the `Image Data` from **Sensors** subsystem form the measured signal, providing information on the current state of vehicle, to the **FCS** subsystem**.**
- The **FCS** subsystem consisting of estimators and controllers computes the command signal to the quadcopter motors, `Actuators,` based on the reference and measured signals. The additional output from **FCS,** `Flag` will stop the simulation if the vehicle state values (position and velocity) cross certain safe limits.
- The **Airframe** subsystem acting as the plant model, takes the `Actuator` commands as inputs to the motors corresponding to the four rotors of the quadcopter. The Multirotor block, the implementation of which is based on [2], and [3], is used in the computation of forces and moments. The output from the **Airframe** subsystem is the `States` bus signal, which is fed back to the **Sensors** subsystem.
- The **Visualization** subsystem uses the `AC cmd` bus signal(reference command signal), the `Actuators` input generated by **FCS,** and the `States` bus signal from **Aiframe** subsystem to aid in the visualization.

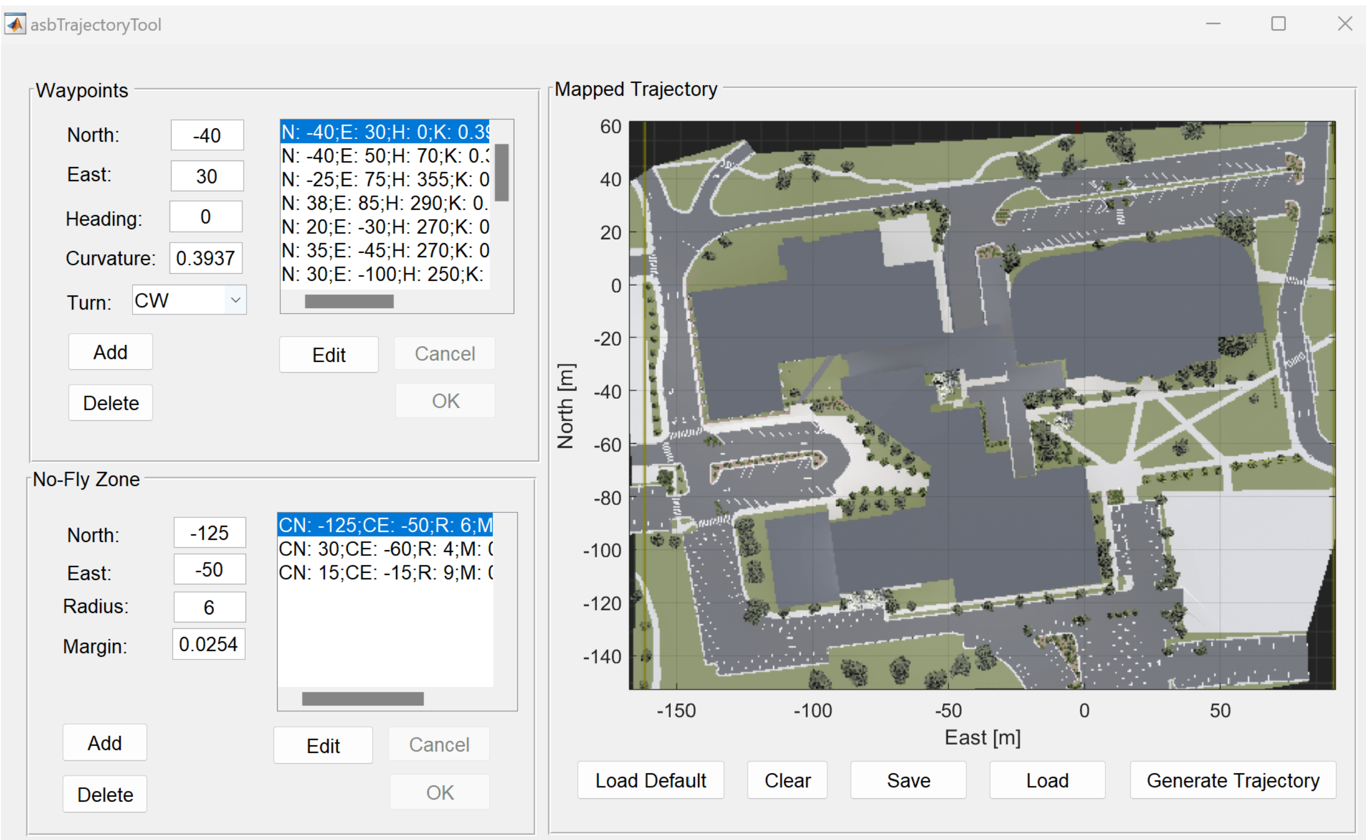
Trajectory Generation

A trajectory generation tool, using the Dubin method, creates a set of navigational waypoints. To create a trajectory with a set of waypoints this method uses a set of poses defined by position, heading, turn curvature, and turn direction.

To start the tool, ensure that the project is open and run:

```
asbTrajectoryTool
```

The following interface displays:



The interface has several panels:

Waypoints

This panel describes the poses the trajectory tool requires. To define these poses, the panel uses text boxes:

- **North** and **East** (position in meters)
- **Heading** (degrees from North)
- **Curvature** (turning curvature in meters^-1)
- **Turn** (direction clockwise or counter-clockwise)

A list of poses appears in the waypoint list to the right of the text boxes.

To add a waypoint, enter pose values in the edit boxes and click **Add**. The new waypoint appears in the waypoint list in the same panel.

To edit the characteristics of a waypoint, select the waypoint in the list and click **Edit**. The characteristics of the waypoints display in the edit boxes. Edit the characteristics as desired, then click **OK**. To cancel the changes click **Cancel**.

To delete a waypoint, in the waypoint list, select the waypoint and click **Delete**.

No-Fly Zone

The panel defines the location and characteristics of the no-fly zones. To define the no-fly zone, the panel uses text boxes:

- **North** and **East** (position in meters)
- **Radius** (distance in meters)
- **Margin** (safety margin in meters)

Use the **Add**, **Delete**, **Edit**, **OK**, and **Cancel** buttons in the same way as for the Waypoints panel.

Mapped Trajectory

This panel plots the trajectory over the Apple Hill campus aerial schematic based on the waypoints and no-fly zone characteristics.

To generate the trajectory, add the waypoint and no-fly zone characteristics to the respective panels, then click **Generate Trajectory**.

To save the trajectory that is currently in your panel, click the **Save** button. This button only saves your last trajectory.

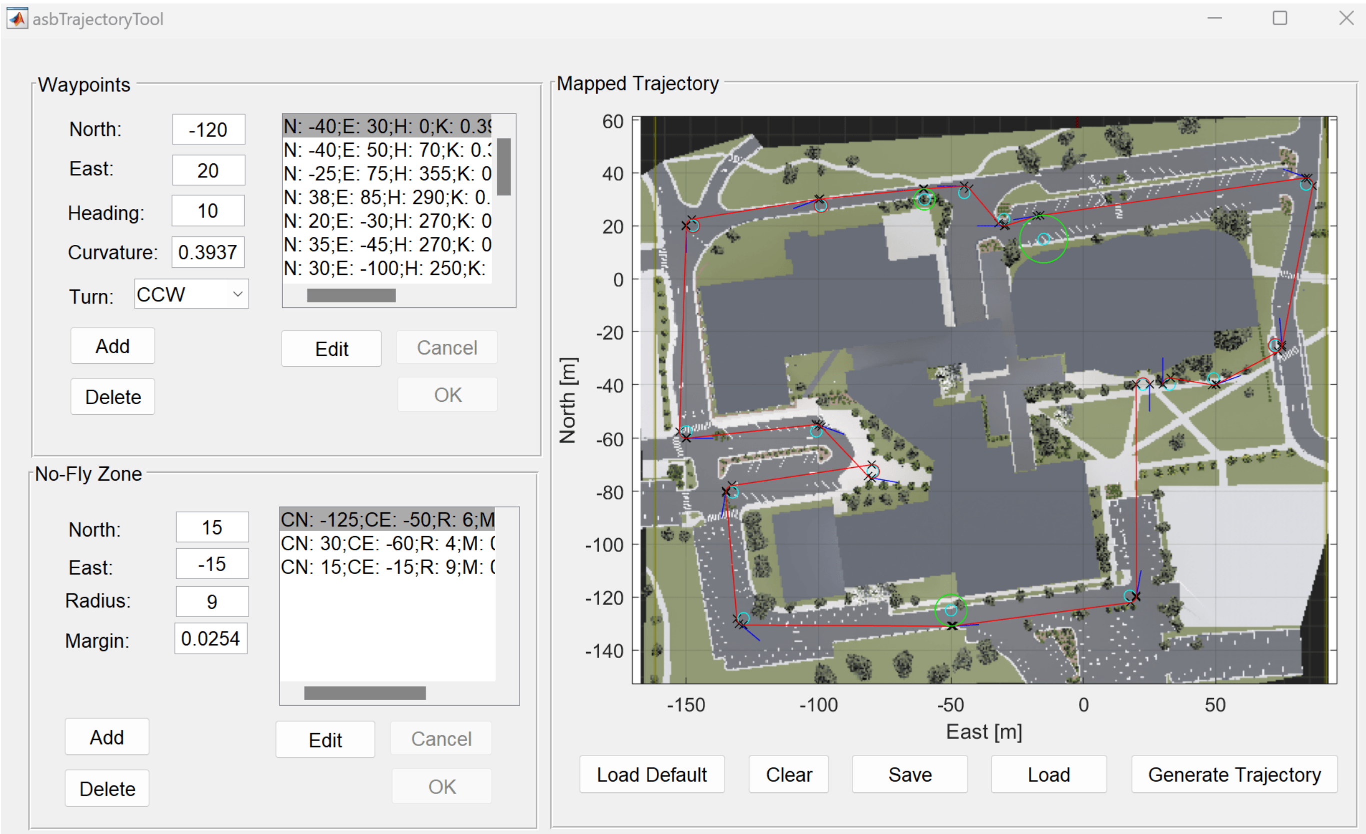
To load the last saved trajectory, click **Load**.

To load the default trajectory, press the **Load Default** button.

To clear the values in the waypoint and no-fly zone panel, click **Clear**.

The default data contains poses for specific locations at which the toy quadcopter uses its cameras so the pilot on the ground can estimate the height of the snow on the roof. Three no-fly zones were defined for each of the auxiliary power generators, so in case there is a failure in the quadcopter, it does not cause any damage to the campus infrastructure.

When the example generates the trajectory for the default data, the plot should appear as follows:



The red line represents the trajectory, black **x** markers determine either a change in the trajectory or a specific pose. Blue lines that represent the heading for that specific waypoint accompany specific poses. No-fly zones are represented as green circles.

References

[1] https://github.com/Parrot-Developers/RollingSpiderEdu/tree/master/MIT_MatlabToolbox/trunk/matlab/libs/RoboticsToolbox.

[2] Prouty, R. Helicopter Performance, Stability, and Control. PWS Publishers, 2005.

[3] Pounds, P., Mahony, R., Corke, P. Modelling and control of a large quadrotor robot. Control Engineering Practice. 2010.

See Also

Related Examples

- “Create Projects”
- “Flight Instrument Gauges” on page 2-58
- “Flight Simulator Interface” on page 2-20

Electrical Component Analysis for Hybrid and Electric Aircraft

This example illustrates how to use modeling for rapid exploration of design space in the hybrid and electric aircraft area and compare the results to design criteria. This process can reduce the number of design iterations and ensure that the final design meets system-level requirements.

Hybrid and electric aircraft are areas of aggressive development in the aerospace industry. To accelerate the process of choosing between hybrid and pure electric power systems, select power network architectures, and size electrical components, consider using simulation with MathWorks® products.

Using preconfigured simulation configurations, this example shows the tradeoffs between battery sizes for a pure electric or hybrid electric power system, with and without payloads. It includes the Pipistrel Alpha Electro and NASA X-57 Maxwell aircraft. Implemented as a project, the example provides various shortcuts that you can use to experiment with simulation.

Model Components

The Aircraft block specifies the aircraft compared in the project:

- Pipistrel Alpha Electro[1], a preconfigured model of the world's first two-seat pure electric training aircraft.
- NASA X-57 Maxwell[2], a preconfigured model of the NASA X-57 Maxwell aircraft, an experimental electric aircraft.
- Custom, an aircraft that you can model according to your specifications.

In the Aircraft block, each aircraft is modeled as a 4th Order Point Mass (Longitudinal) in flight, with the required thrust output as a load on the motor. This abstract model assumes that the pilot takes the actions necessary to follow the mission.

You can specify several aerodynamic characteristics for the selected aircraft, including empty and maximum mass, wing area and lift curve values, a drag coefficient, and target speeds for the climb, cruise, and descent portions of the mission. The block uses these values, along with the mission altitudes, to create lookup tables for angle of attack (alpha) and thrust, given atmospheric density, target speed, and flight profile angle (gamma). At every point along the flight, the lookup tables return alpha and thrust for input into the 4th-order Point Mass block, which calculates the accelerations. By holding alpha and thrust at the calculated, steady-state values, the actual speed quickly reaches the desired speed. The Aircraft block also defines the climb and descent rates for the mission.

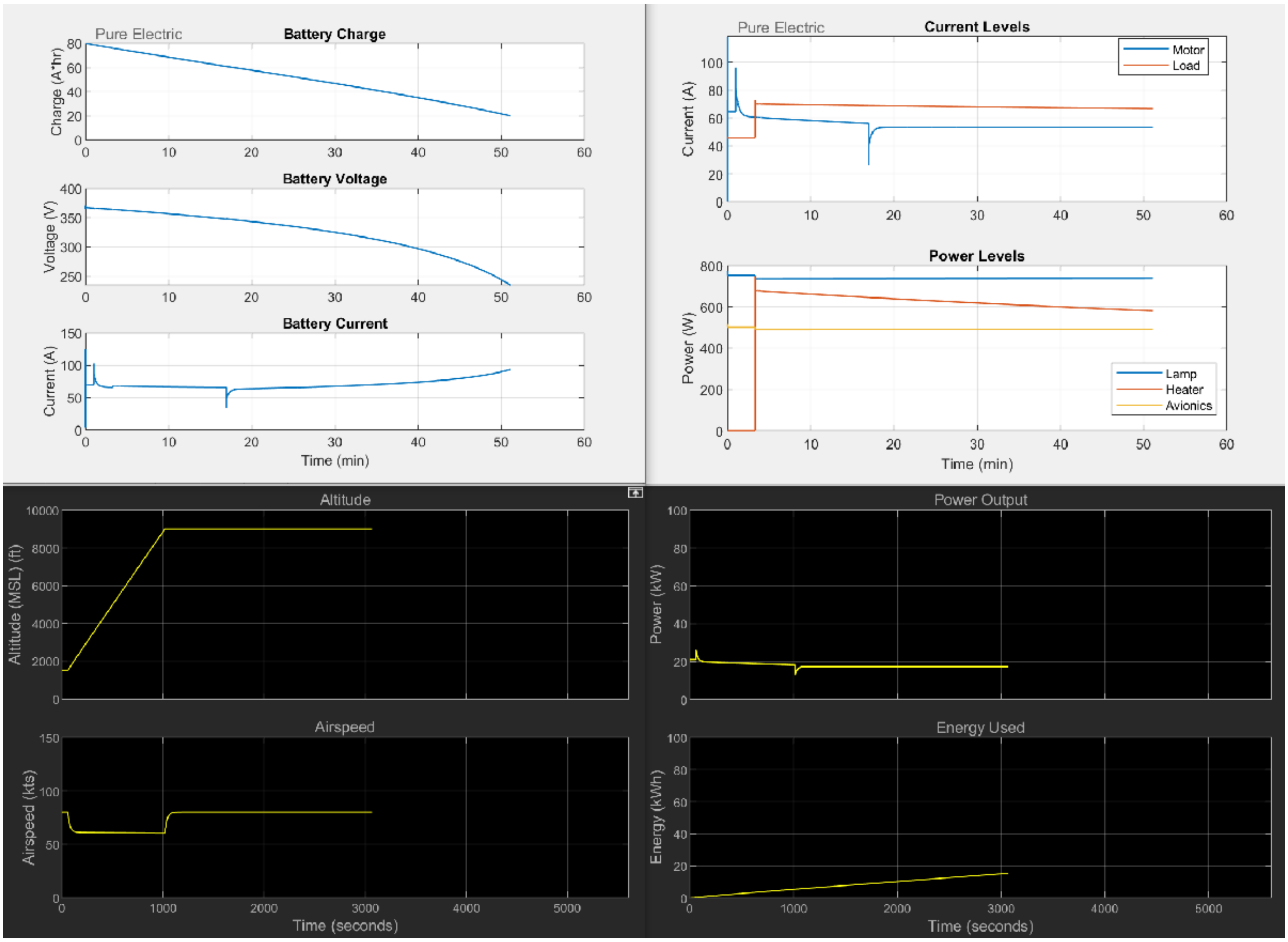
The Mission Profile block sets the airport and cruise altitudes and the total flight distance. If entered values are not feasible, such as a distance too short for the aircraft to climb to and descend from the requested cruise altitude, then the values are adjusted, and a message describes the change. To see how far the aircraft can fly until it runs out of battery power, enter a long total flight distance.

The Environment block calculates the air density at the chosen altitudes using the COESA Atmosphere Model.

Run the Model with Default Settings

By default, the Aircraft block is configured for the Pipistrel Alpha Electro aircraft. To see the performance of this aircraft with default settings, run the model using the "Single Run" project shortcut. The Pipistrel is run with no payload.

Two figures display showing the battery states, current, and power levels during the flight. Two scope windows show the mission progress (altitude and airspeed), power output, and energy used. Note, the battery runs out of capacity (reaches 20 amp-hours) before the entire mission is completed.



To capture the data created by a simulation, the example uses the Simscape “Data Logging” (Simscape) capability. The various simulation cases provided by the project shortcuts run scripts that run the desired cases and then extract the results from the Simscape log to create the figures. For aircraft with more than one electric motor, the total required torque calculated by the Aircraft subsystem is divided by the number of motors before being passed to the Power subsystem. The criteria to stop the simulation, `batteryCapacityMin`, is adjusted up from 20 amp-hours accordingly.

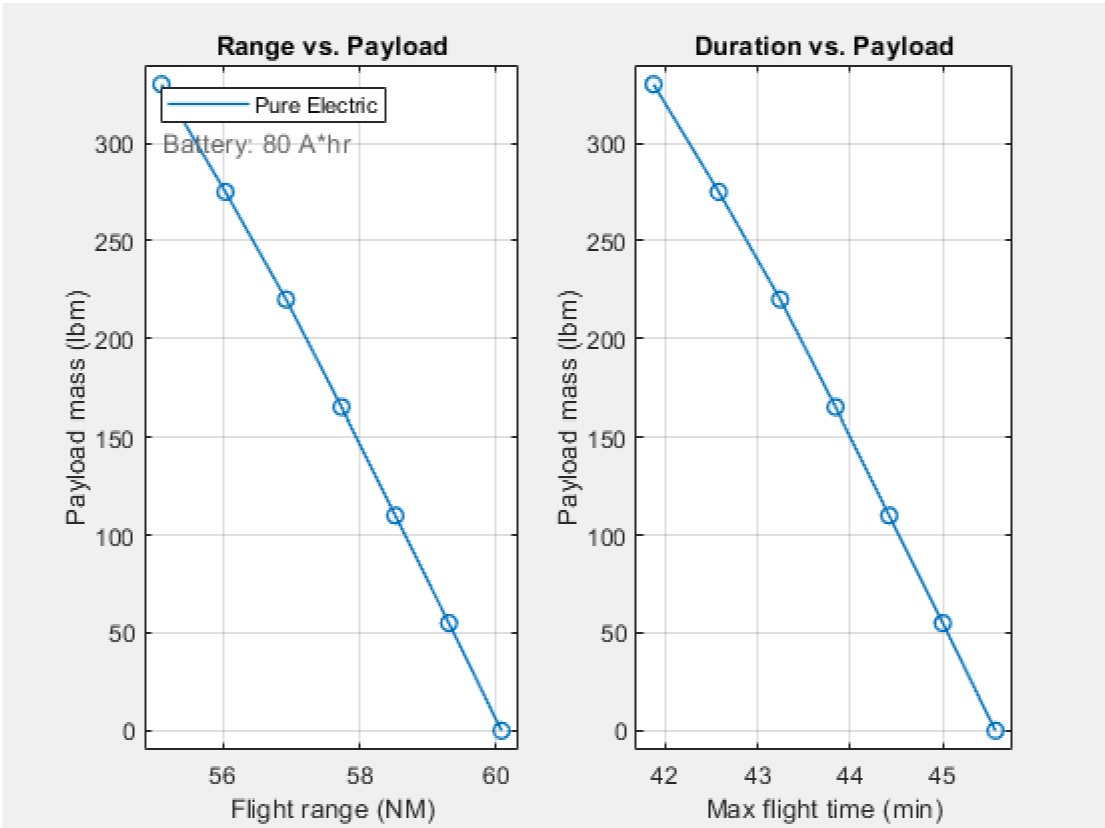
Run the Model with Cruise at 3000 Feet

The Pipistrel is designed as a primary flight trainer. The default mission configured for the model might not be the typical mission for the Pipistrel. Modify the mission to cruise at 3000 instead of 9000 feet, and then make a single run to see the effect of this change (duration decreases).

Run the Model with a Payload

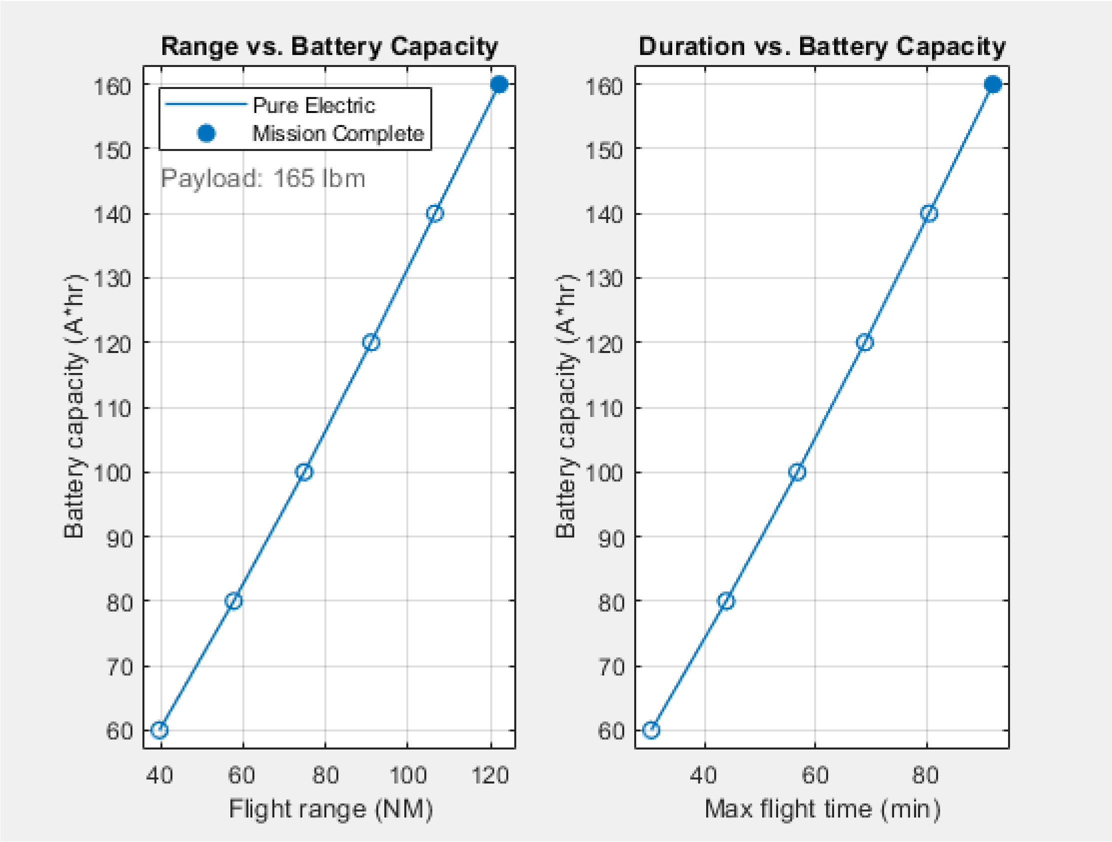
To run the Pipistrel with a 165 pound payload, use the "Set Payload Mass" shortcut, then run the model again. To see the effect of a range of payload values, use the "Sweep Payload Mass" shortcut.

This shortcut varies the payload from 0 to 330 pounds. The sweeps produce different figures, showing the flight time and range for the swept parameter. Each marker represents one simulation. Hover your mouse over a marker to see its payload mass ("X") and flight range or duration ("Y") values.



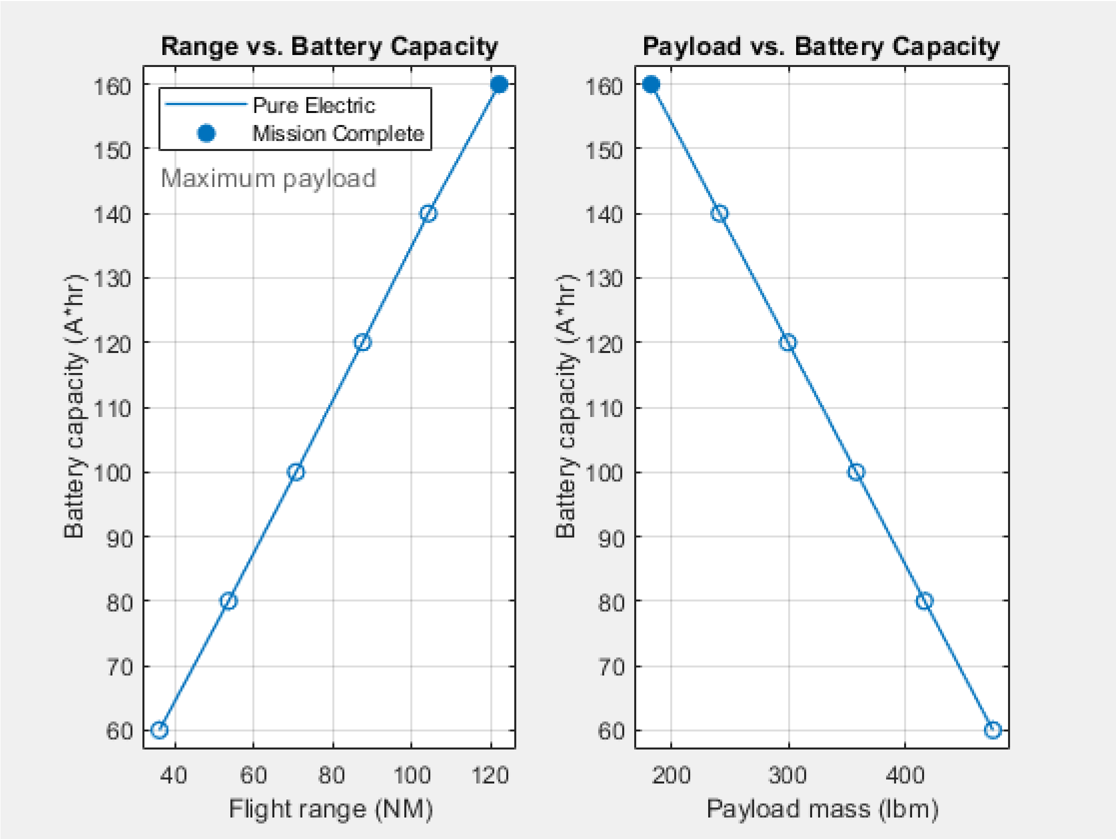
Run the Model with Various Battery Sizes

To see how the flight range is affected by the battery size, use the "Sweep Battery Size" shortcut. The shortcut varies the capacity from 60 to 160 amp-hours (or 100 to 200 amp-hours if the X-57 is selected). The example assumes the battery mass to be linearly proportional to its capacity, so increasing its capacity increases its mass as well. If the payload is set large enough (over 183 pounds for the Pipistrel), this increase in battery mass can cause the largest battery in the sweep to put the aircraft over its maximum mass value (e.g. 1212 pounds for the Pipistrel). The `total_mass` variable in the base workspace stores the total mass of each case in the sweep. If the battery capacity is enough to complete the mission, a filled marker indicates this. Note that pure electric aircraft, such as the Pipistrel, have no fuel burn, resulting in no change in mass throughout a flight.



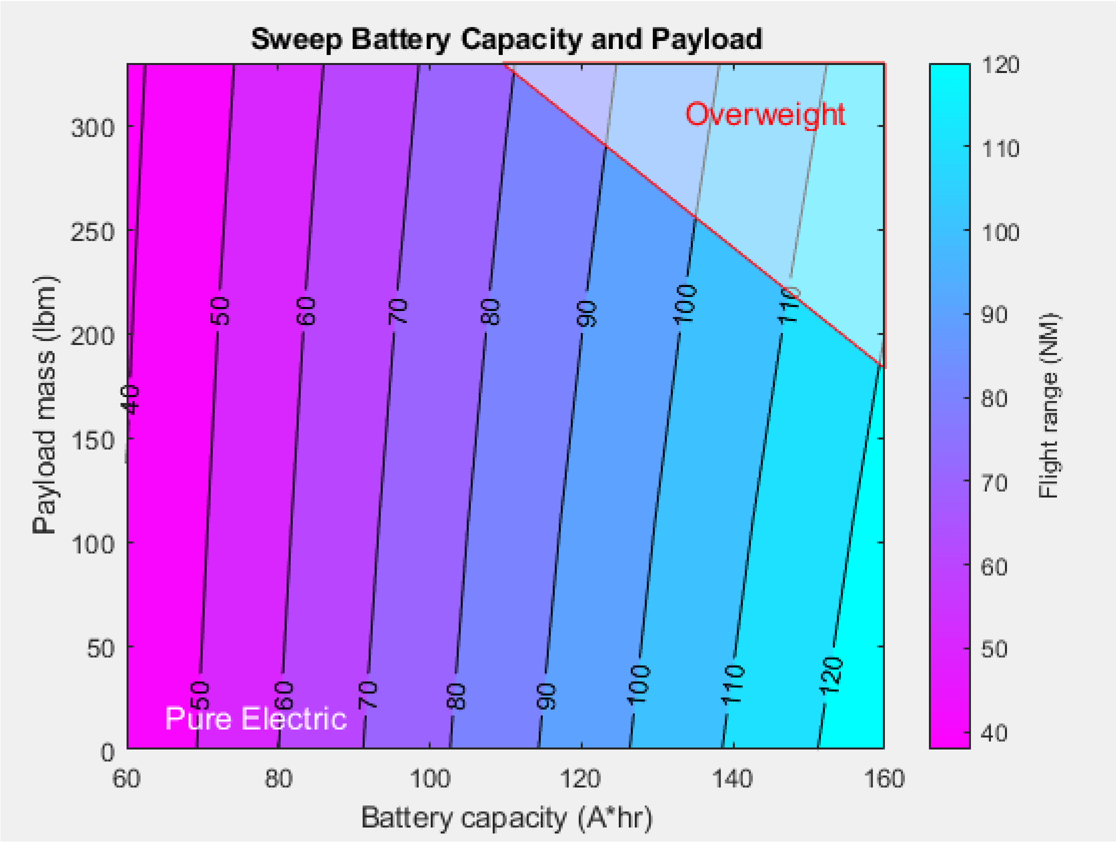
Run the Model with Maximum Payload

The battery capacity sweep in "Run the Model with Various Battery Sizes" yields the flight range for a fixed payload. To find the flight range for the maximum payload, use the "Sweep Range at Max Payload" shortcut. This sweeps the battery capacity with the payload set such that the total mass equals the maximum mass in every case (unless payload becomes negative, in which case payload is set to zero and the model goes overweight). From these results, you can choose the maximum battery size for a given payload requirement.



Run the Model with Various Battery and Payload Sizes

To simultaneously see the flight range distance dependence on both battery size and payload mass, use the "Sweep Battery & Payload" shortcut, which produces a contour plot. Areas where the aircraft is over the maximum weight are denoted with a red and white "Overweight" overlay.



Run the Model with the Hybrid Electric Option

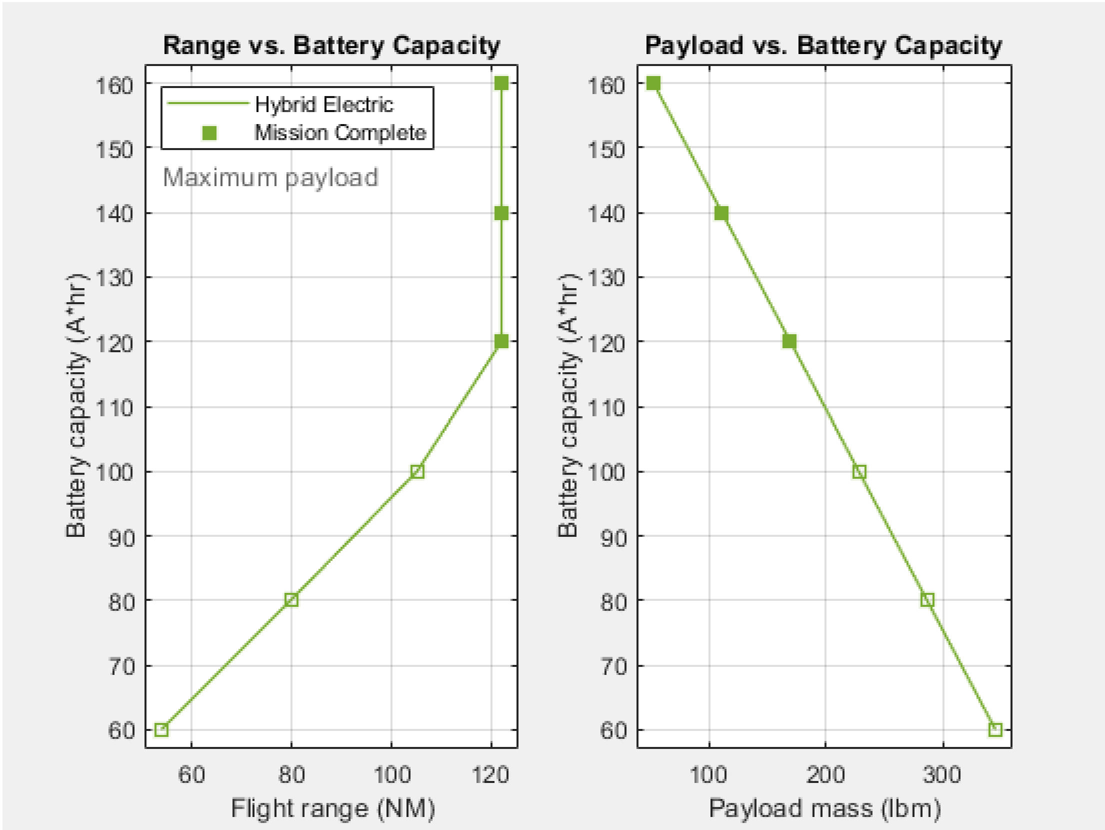
Since battery power density is much less than that of aviation fuel, pure electric aircraft have less distance range than their fuel-powered counterparts. To bridge this gap, consider a hybrid electric power system. To recharge the batteries, the hybrid power subsystem variant in this example adds a 130 pound, 50 kW, two-stroke piston engine and generator to the pure electric power subsystem components.

To try a hybrid power system, perform one of these workflows:

Change Power Subsystem Variant and Run the Battery Range Sweep

1. Use the "Hybrid Electric" shortcut. This shortcut changes the Power subsystem variant.

2. To repeat the sweep previously done for pure electric power, use "Sweep Range at Max Payload".

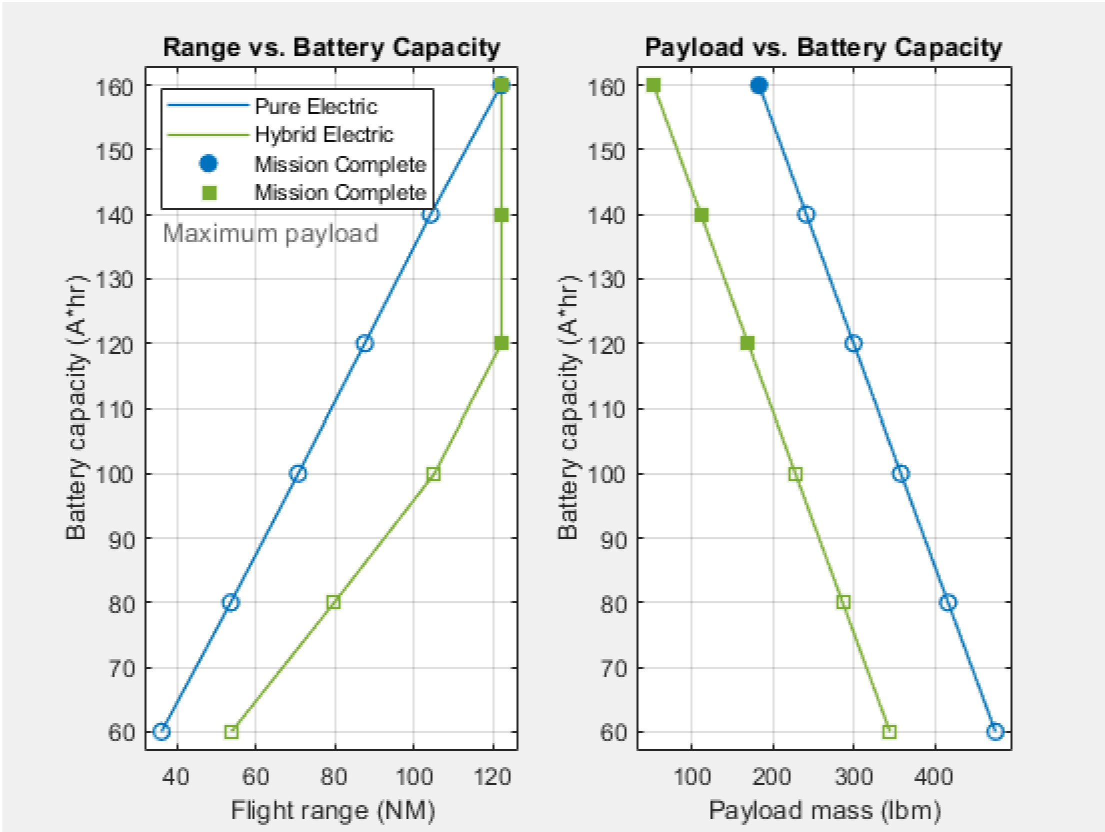


In this case the larger battery sizes are capable of longer missions than what is currently defined (120 NM). Try entering a longer total flight distance (e.g. 200 NM) in the Mission Profile and then rerun this sweep.

Run the Battery Range Sweep for Both Power Variants

1. Use the "Hybrid/Electric Range Comparison" shortcut to run the sweep for both power variants.

2. Compare the results in a single figure. The results show that a hybrid power system can improve range, but at the expense of payload.



Explore How the Mission Affects Range and Endurance

To explore how the mission affects the range and endurance, select the Custom aircraft model in the Aircraft block. The default configuration values for this aircraft are the same as for the Pipistrel, with the exception of the speeds. Adjust the speeds and mission altitudes as desired, then run and compare the results to those for the Pipistrel with default settings.

If the custom aircraft being evaluated is significantly different from the Pipistrel, then accordingly adjust the `CustomAircraft` values in the "asbHybridAircraftDefaults.m" file.

Additional Model Details

Power Subsystem

The power subsystem is modeled with two variant models: Pure Electric and Hybrid Electric, controlled by the variable `POWER_MODE` in the base workspace.

The Pure Electric model includes a battery, high- and low-voltage DC networks, and a mechanical model of the aircraft. The mechanical model acts as a load on the high-voltage DC network. The low-voltage DC network includes a set of loads that turn on and off during the flight mission.

The series Hybrid Electric model contains all the components in the Pure Electric model, plus a 50 kW engine, a generator, and fuel. The Generic Engine (Simscape Driveline) drives a generator that supplements the power available from the battery. The generator recharges the battery during flight. The mass of the fuel consumed by the engine is included in the simulation. The low-voltage DC network includes a set of loads that turn on and off during the flight cycle, including the fuel pump for the combustion engine.

These two variant models are composed of three or four subsystems for load torque, the motor, the generator, and the DC power distribution.

Load Torque Subsystem

This subsystem converts the required mechanical power into the load torque on the motor shaft. This model assumes that a specified amount of the motor mechanical power is converted into thrust. Dividing the required power to maintain thrust by the motor speed results in the load torque on the motor shaft. The motor control system adjusts to maintain the required shaft speed under the varying load.

Motor Subsystem

This subsystem represents the electric motor and drive electronics operating in torque-control mode, or equivalently, current-control mode. The motor permissible range of torques and speeds is defined by a torque-speed envelope.

Fuel Pump Subsystem

This subsystem models the fuel pump. An electric motor drives a pump that pushes fuel through a valve. The opening of the valve varies during the flight, which changes the current that the motor draws from the DC network.

Generator Subsystem

This subsystem represents the generator and drive electronics operating in torque-control mode, or equivalently, current-control mode. It is driven by the combustion engine to supply additional electrical power to the aircraft network.

DC Power Distribution Subsystem

This subsystem models the breakers that open and close to connect and disconnect loads from the low-voltage DC network. The varying conditions affect the power drawn from the network, the range of the aircraft, and the power requirements for the power lines in the aircraft.

References

[1] https://www.pipistrel-aircraft.com/products/velis-electro/

[2] https://www.nasa.gov/aeronautics/x-57-maxwell/

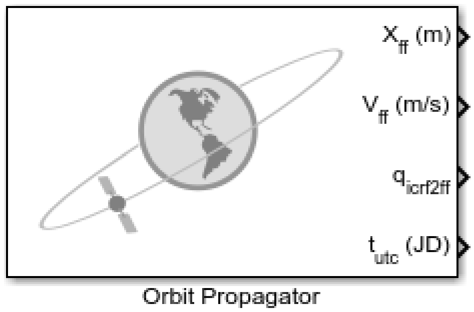
See Also

4th Order Point Mass (Longitudinal) | COESA Atmosphere Model

Constellation Modeling with the Orbit Propagator Block

This example shows how to propagate the orbits of a constellation of satellites and compute and visualize access intervals between the individual satellites and several ground stations. It uses:

- Aerospace Blockset™ **Orbit Propagator** block
- Aerospace Toolbox **satelliteScenario** object



The Aerospace Toolbox **satelliteScenario** object lets you load previously generated, time-stamped ephemeris data into a scenario from a timeseries or timetable object. Data is interpolated in the scenario object to align with the scenario time steps, allowing you to incorporate data generated in a Simulink® model into either a new or existing **satelliteScenario** object. This example shows how to propagate a constellation of satellites in Simulink with the Aerospace Blockset **Orbit Propagator** block, and load the logged ephemeris data into a **satelliteScenario** object for access analysis.

Define Mission Parameters and Constellation Initial Conditions

Specify a start date and duration for the mission. This example uses MATLAB® structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

```
mission.StartDate = datetime(2020, 11, 30, 22, 23, 24);
mission.Duration  = hours(24);
```

The constellation in this example is a Walker-Delta constellation modeled similar to Galileo, the European GNSS (Global navigation satellite system) constellation. The constellation consists of 24 satellites in medium Earth orbit (MEO).

Walker-Delta constellations use the notation:

$$i{:}T/P/F$$

where

i = inclination

T = total number of satellites

P = number of equally space geometric planes

F = spacing between satellites in adjacent planes

Walker-Delta constellations are a common solution for maximizing geometric coverage over Earth while minimizing the number of satellites required to perform the mission. The Galileo navigation system is a Walker-Delta 56°:24/3/1 constellation (24 satellites in 3 planes inclined at 56 degrees) in a 29599.8 km orbit.

Specify Keplerian orbital elements for the constellation at `mission.StartDate`.

```
mission.Satellites.SemiMajorAxis  = 29599.8e3 * ones(24,1); % meters
mission.Satellites.Eccentricity   = 0.0005    * ones(24,1);
mission.Satellites.Inclination    = 56        * ones(24,1); % deg
mission.Satellites.ArgOfPeriapsis = 350       * ones(24,1); % deg
```

The ascending nodes of the orbital planes of a Walker-Delta constellation are uniformly distributed at intervals of $\frac{360^{\circ}}{P}$ around the equator. The number of satellites per plane, S, is given as $S = \frac{T}{P}$. With 24 satellites total, this results in 3 planes of 8 satellites at 120 degree intervals around the equator. The satellites in each orbital plane are distributed at intervals of $\frac{360^{\circ}}{S}$, or 45 degrees.

```
mission.Satellites.RAAN                = sort(repmat([0 120 240], 1,8))'; % deg
mission.Satellites.TrueAnomaly         = repmat(0:45:315, 1,3)'; % deg
```

Lastly, account for the relative angular shift between adjacent orbital planes. The phase difference is given as $\Delta\phi = F * \frac{360}{T}$, or 15 degrees in this case.

```
mission.Satellites.TrueAnomaly(9:16)  = mission.Satellites.TrueAnomaly(9:16)  + 15;
mission.Satellites.TrueAnomaly(17:24) = mission.Satellites.TrueAnomaly(17:24) + 30;
```

Show the constellation nodes in a table.

```
ConstellationDefinition = table(mission.Satellites.SemiMajorAxis, ...
    mission.Satellites.Eccentricity, ...
    mission.Satellites.Inclination, ...
    mission.Satellites.RAAN, ...
    mission.Satellites.ArgOfPeriapsis, ...
    mission.Satellites.TrueAnomaly, ...
    'VariableNames', ["a (m)", "e", "i (deg)", "Ω (deg)", "ω (deg)", "ν (deg)"])
```

ConstellationDefinition=*24×6 table*

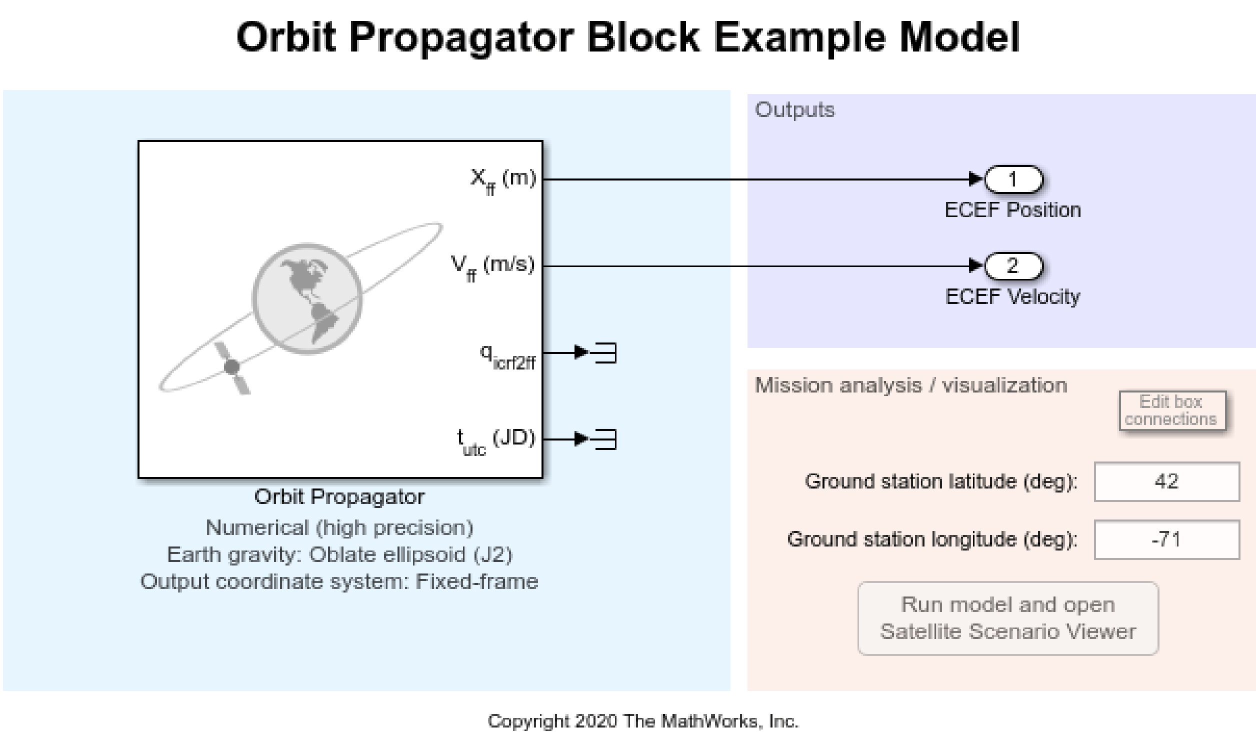
a (m)	e	i (deg)	Ω (deg)	ω (deg)	ν (deg)
2.96e+07	0.0005	56	0	350	0
2.96e+07	0.0005	56	0	350	45
2.96e+07	0.0005	56	0	350	90
2.96e+07	0.0005	56	0	350	135
2.96e+07	0.0005	56	0	350	180
2.96e+07	0.0005	56	0	350	225
2.96e+07	0.0005	56	0	350	270
2.96e+07	0.0005	56	0	350	315
2.96e+07	0.0005	56	120	350	15
2.96e+07	0.0005	56	120	350	60
2.96e+07	0.0005	56	120	350	105
2.96e+07	0.0005	56	120	350	150
2.96e+07	0.0005	56	120	350	195
2.96e+07	0.0005	56	120	350	240
2.96e+07	0.0005	56	120	350	285

```
    2.96e+07     0.0005       56          120          350          330
      ⋮
```

Open and Configure the Orbit Propagation Model

Open the included Simulink model. This model contains an **`Orbit Propagator`** block connected to output ports. The **`Orbit Propagator`** block supports vectorization. This allows you to model multiple satellites in a single block by specifying arrays of initial conditions in the `Block Parameters` window or using `set_param`. The model also includes a "Mission Analysis and Visualization" section that contains a dashboard **`Callback button`**. When clicked, this button runs the model, creates a new satelliteScenario object in the global base workspace containing the satellite or constellation defined in the **`Orbit Propagator`** block, and opens a Satellite Scenario Viewer window for the new scenario. To view the source code for this action, double click the callback button. **The "Mission Analysis and Visualization" section is a standalone workflow to create a new satelliteScenario object and is not used as part of this written example.**

```
mission.mdl = "OrbitPropagatorBlockExampleModel";
open_system(mission.mdl);
```



Define the path to the **`Orbit Propagator`** block in the model.

```
mission.Satellites.blk = mission.mdl + "/Orbit Propagator";
```

Set satellite initial conditions. To assign the Keplerian orbital element set defined in the previous section, use `set_param`.

```
set_param(mission.Satellites.blk, ...
    startDate = num2str(juliandate(mission.StartDate)), ...
    stateFormatNum = "Orbital elements", ...
```

```
    orbitType      = "Keplerian", ...
    semiMajorAxis  = "mission.Satellites.SemiMajorAxis", ...
    eccentricity   = "mission.Satellites.Eccentricity", ...
    inclination    = "mission.Satellites.Inclination", ...
    raan           = "mission.Satellites.RAAN", ...
    argPeriapsis   = "mission.Satellites.ArgOfPeriapsis", ...
    trueAnomaly    = "mission.Satellites.TrueAnomaly");
```

Set the position and velocity output ports of the block to use the Earth-centered Earth-fixed frame, which is the International Terrestrial Reference Frame (ITRF).

```
set_param(mission.Satellites.blk, ...
    centralBody  = "Earth", ...
    outportFrame = "Fixed-frame");
```

Configure the propagator. This example uses the `Oblate ellipsoid (J2)` propagator which includes second order zonal harmonic perturbations in the satellite trajectory calculations, accounting for the oblateness of Earth.

```
set_param(mission.Satellites.blk, ...
    propagator   = "Numerical (high precision)", ...
    gravityModel = "Oblate ellipsoid (J2)", ...
    useEOPs      = "off");
```

Apply model-level solver setting using `set_param`. For best performance and accuracy when using a numerical propagator, use a variable-step solver.

```
set_param(mission.mdl, ...
    SolverType = "Variable-step", ...
    SolverName = "VariableStepAuto", ...
    RelTol     = "1e-6", ...
    AbsTol     = "1e-7", ...
    StopTime   = string(seconds(mission.Duration)));
```

Save model output port data as a dataset of time series objects.

```
set_param(mission.mdl, ...
    SaveOutput = "on", ...
    OutputSaveName = "yout", ...
    SaveFormat = "Dataset");
```

Run the Model and Collect Satellite Ephemerides

Simulate the model.

```
mission.SimOutput = sim(mission.mdl);
```

Extract position and velocity data from the model output data structure.

```
mission.Satellites.TimeseriesPosECEF = mission.SimOutput.yout{1}.Values;
mission.Satellites.TimeseriesVelECEF = mission.SimOutput.yout{2}.Values;
```

Set the start data from the mission in the timeseries object.

```
mission.Satellites.TimeseriesPosECEF.TimeInfo.StartDate = mission.StartDate;
mission.Satellites.TimeseriesVelECEF.TimeInfo.StartDate = mission.StartDate;
```

The timeseries objects contain position and velocity data for all 24 satellites.

```
mission.Satellites.TimeseriesPosECEF
```

```
  timeseries

  Common Properties:
            Name: ''
            Time: [57x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
            Data: [24x3x57 double]
        DataInfo: [1x1 tsdata.datametadata]

  More properties, Methods
```

Load the Satellite Ephemerides into a `satelliteScenario` Object

Create a satellite scenario object for the analysis.

```
scenario = satelliteScenario(mission.StartDate, mission.StartDate + hours(24), 60);
```

Add all 24 satellites to the satellite scenario from the ECEF position and velocity timeseries objects using the `satellite` method.

```
sat = satellite(scenario, mission.Satellites.TimeseriesPosECEF, mission.Satellites.TimeseriesVelE
    CoordinateFrame="ecef", Name="GALILEO " + (1:24))
```

```
sat =
  1x24 Satellite array with properties:

    Name
    ID
    ConicalSensors
    Gimbals
    Transmitters
    Receivers
    Accesses
    Eclipse
    GroundTrack
    Orbit
    CoordinateAxes
    OrbitPropagator
    MarkerColor
    MarkerSize
    ShowLabel
    LabelFontColor
    LabelFontSize
    Visual3DModel
    Visual3DModelScale
```

```
disp(scenario)
```

```
  satelliteScenario with properties:

         StartTime: 30-Nov-2020 22:23:24
          StopTime: 01-Dec-2020 22:23:24
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×24 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
```

```
     Viewers: [0×0 matlabshared.satellitescenario.Viewer]
    AutoShow: 1
```

Set Graphical Properties on the Satellites

Set satellites in each orbital plane to have the same orbit color.

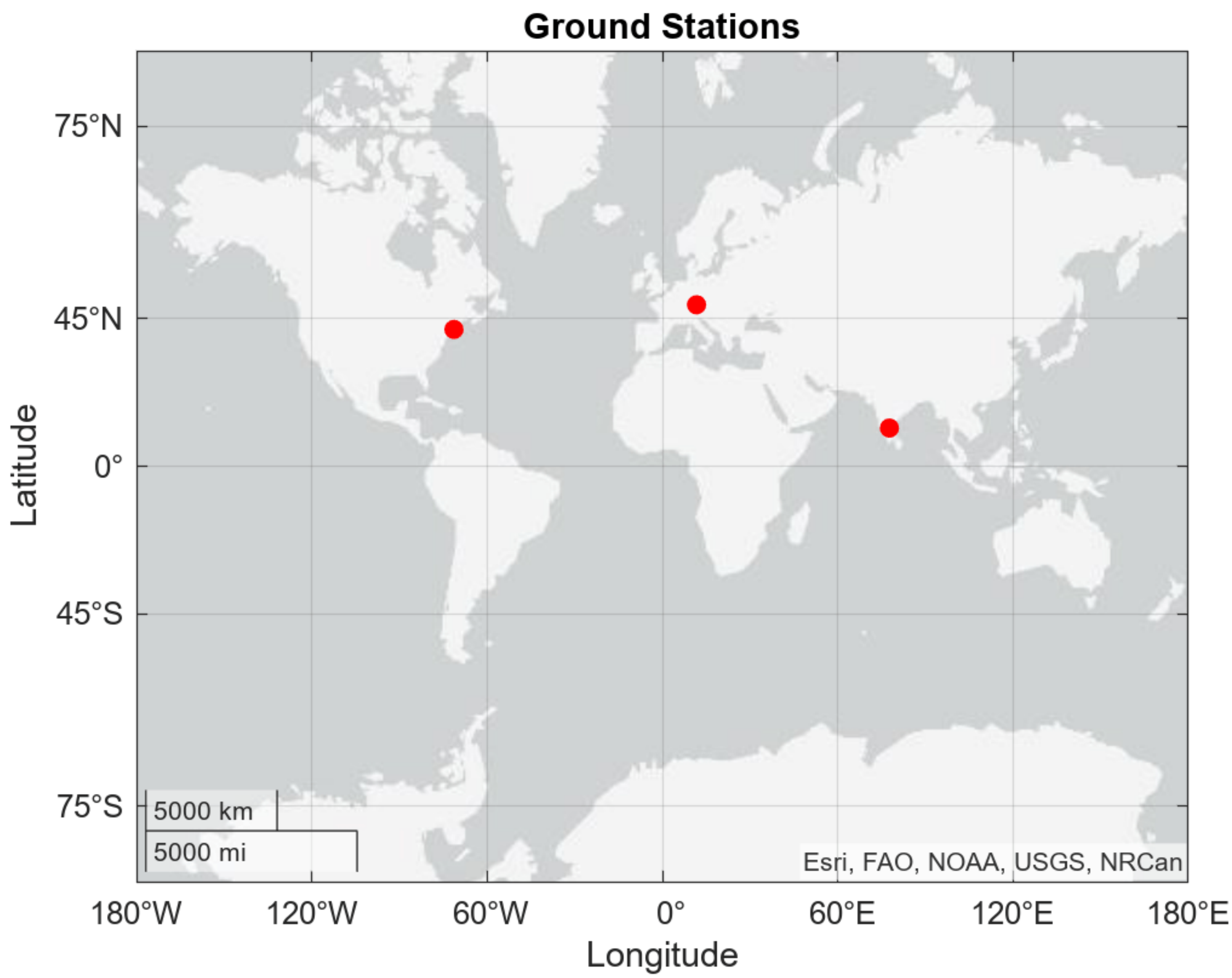
```
set(sat(1:8), MarkerColor="#FF6929");
set(sat(9:16), MarkerColor="#139FFF");
set(sat(17:24), MarkerColor="#64D413");
orbit = [sat(:).Orbit];
set(orbit(1:8), LineColor="#FF6929");
set(orbit(9:16), LineColor="#139FFF");
set(orbit(17:24), LineColor="#64D413");
```

Add Ground Stations to Scenario

To provide accurate positioning data, a location on Earth must have access to at least 4 satellites in the constellation at any given time. In this example, use three MathWorks® locations to compare total constellation access over the 1 day analysis window to different regions of Earth:

- Natick, Massachusetts, USA (42.30048°, -71.34908°)
- München, Germany (48.23206°, 11.68445°)
- Bangalore, India (12.94448°, 77.69256°)

```
gsUS = groundStation(scenario, 42.30048, -71.34908, ...
    MinElevationAngle=10, Name="Natick");
gsUS.MarkerColor = "red";
gsDE = groundStation(scenario, 48.23206, 11.68445, ...
    MinElevationAngle=10, Name="Munchen");
gsDE.MarkerColor = "red";
gsIN = groundStation(scenario, 12.94448, 77.69256, ...
    MinElevationAngle=10, Name="Bangalore");
gsIN.MarkerColor = "red";


figure
geoscatter([gsUS.Latitude gsDE.Latitude gsIN.Latitude], ...
    [gsUS.Longitude gsDE.Longitude gsIN.Longitude], "red", "filled")
geolimits([-75 75], [-180 180])
title("Ground Stations")
```



Compute Ground Station to Satellite Access (Line-of-Sight Visibility)

Calculate line-of-sight access between the ground stations and each individual satellite using the `access` method.

```
accessUS = access(gsUS, sat);
accessDE = access(gsDE, sat);
accessIN = access(gsIN, sat);
```

Set access colors to match orbital plane colors assigned earlier in the example.

```
set(accessUS, LineWidth="1");
set(accessUS(1:8), LineColor="#FF6929");
set(accessUS(9:16), LineColor="#139FFF");
set(accessUS(17:24), LineColor="#64D413");

set(accessDE, LineWidth="1");
set(accessDE(1:8), LineColor="#FF6929");
set(accessDE(9:16), LineColor="#139FFF");
set(accessDE(17:24), LineColor="#64D413");
```

```
set(accessIN, LineWidth="1");
set(accessIN(1:8), LineColor="#FF6929");
set(accessIN(9:16), LineColor="#139FFF");
set(accessIN(17:24), LineColor="#64D413");
```

View the full access table between each ground station and all satellites in the constellation as tables. Sort the access intervals by interval start time. Satellites added from ephemeris data do not display values for StartOrbit and Stop orbit.

```
intervalsUS = accessIntervals(accessUS);
intervalsUS = sortrows(intervalsUS, "StartTime", "ascend")
```

```
intervalsUS=40×8 table
     Source        Target       IntervalNumber         StartTime                EndTime
    ________    ____________    ______________    ____________________    ____________________

    "Natick"    "GALILEO 1"           1           30-Nov-2020 22:23:24    01-Dec-2020 04:04:24
    "Natick"    "GALILEO 2"           1           30-Nov-2020 22:23:24    01-Dec-2020 01:24:24
    "Natick"    "GALILEO 3"           1           30-Nov-2020 22:23:24    30-Nov-2020 22:57:24
    "Natick"    "GALILEO 12"          1           30-Nov-2020 22:23:24    01-Dec-2020 00:00:24
    "Natick"    "GALILEO 13"          1           30-Nov-2020 22:23:24    30-Nov-2020 23:05:24
    "Natick"    "GALILEO 18"          1           30-Nov-2020 22:23:24    01-Dec-2020 04:00:24
    "Natick"    "GALILEO 19"          1           30-Nov-2020 22:23:24    01-Dec-2020 01:42:24
    "Natick"    "GALILEO 20"          1           30-Nov-2020 22:23:24    30-Nov-2020 22:46:24
    "Natick"    "GALILEO 11"          1           30-Nov-2020 22:25:24    01-Dec-2020 00:18:24
    "Natick"    "GALILEO 17"          1           30-Nov-2020 22:50:24    01-Dec-2020 05:50:24
    "Natick"    "GALILEO 8"           1           30-Nov-2020 23:20:24    01-Dec-2020 07:09:24
    "Natick"    "GALILEO 7"           1           01-Dec-2020 01:26:24    01-Dec-2020 10:00:24
    "Natick"    "GALILEO 24"          1           01-Dec-2020 01:40:24    01-Dec-2020 07:12:24
    "Natick"    "GALILEO 14"          1           01-Dec-2020 03:56:24    01-Dec-2020 07:15:24
    "Natick"    "GALILEO 6"           1           01-Dec-2020 04:05:24    01-Dec-2020 12:14:24
    "Natick"    "GALILEO 23"          1           01-Dec-2020 04:10:24    01-Dec-2020 08:03:24
       ⋮
```

```
intervalsDE = accessIntervals(accessDE);
intervalsDE = sortrows(intervalsDE, "StartTime", "ascend")
```

```
intervalsDE=40×8 table
     Source         Target       IntervalNumber         StartTime                EndTime
    _________    ____________    ______________    ____________________    ____________________

    "Munchen"    "GALILEO 2"           1           30-Nov-2020 22:23:24    01-Dec-2020 04:34:24
    "Munchen"    "GALILEO 3"           1           30-Nov-2020 22:23:24    01-Dec-2020 01:58:24
    "Munchen"    "GALILEO 4"           1           30-Nov-2020 22:23:24    30-Nov-2020 23:05:24
    "Munchen"    "GALILEO 10"          1           30-Nov-2020 22:23:24    30-Nov-2020 23:58:24
    "Munchen"    "GALILEO 19"          1           30-Nov-2020 22:23:24    01-Dec-2020 01:36:24
    "Munchen"    "GALILEO 20"          1           30-Nov-2020 22:23:24    01-Dec-2020 00:15:24
    "Munchen"    "GALILEO 21"          1           30-Nov-2020 22:23:24    30-Nov-2020 22:28:24
    "Munchen"    "GALILEO 9"           1           30-Nov-2020 22:34:24    01-Dec-2020 02:22:24
    "Munchen"    "GALILEO 18"          1           30-Nov-2020 22:41:24    01-Dec-2020 02:31:24
    "Munchen"    "GALILEO 1"           1           30-Nov-2020 23:05:24    01-Dec-2020 06:42:24
    "Munchen"    "GALILEO 16"          1           30-Nov-2020 23:29:24    01-Dec-2020 04:47:24
    "Munchen"    "GALILEO 15"          1           01-Dec-2020 00:50:24    01-Dec-2020 07:27:24
    "Munchen"    "GALILEO 17"          1           01-Dec-2020 01:05:24    01-Dec-2020 03:00:24
    "Munchen"    "GALILEO 8"           1           01-Dec-2020 01:57:24    01-Dec-2020 08:25:24
    "Munchen"    "GALILEO 14"          1           01-Dec-2020 02:36:24    01-Dec-2020 10:19:24
    "Munchen"    "GALILEO 7"           1           01-Dec-2020 04:35:24    01-Dec-2020 09:43:24
```

```
    ⋮
```

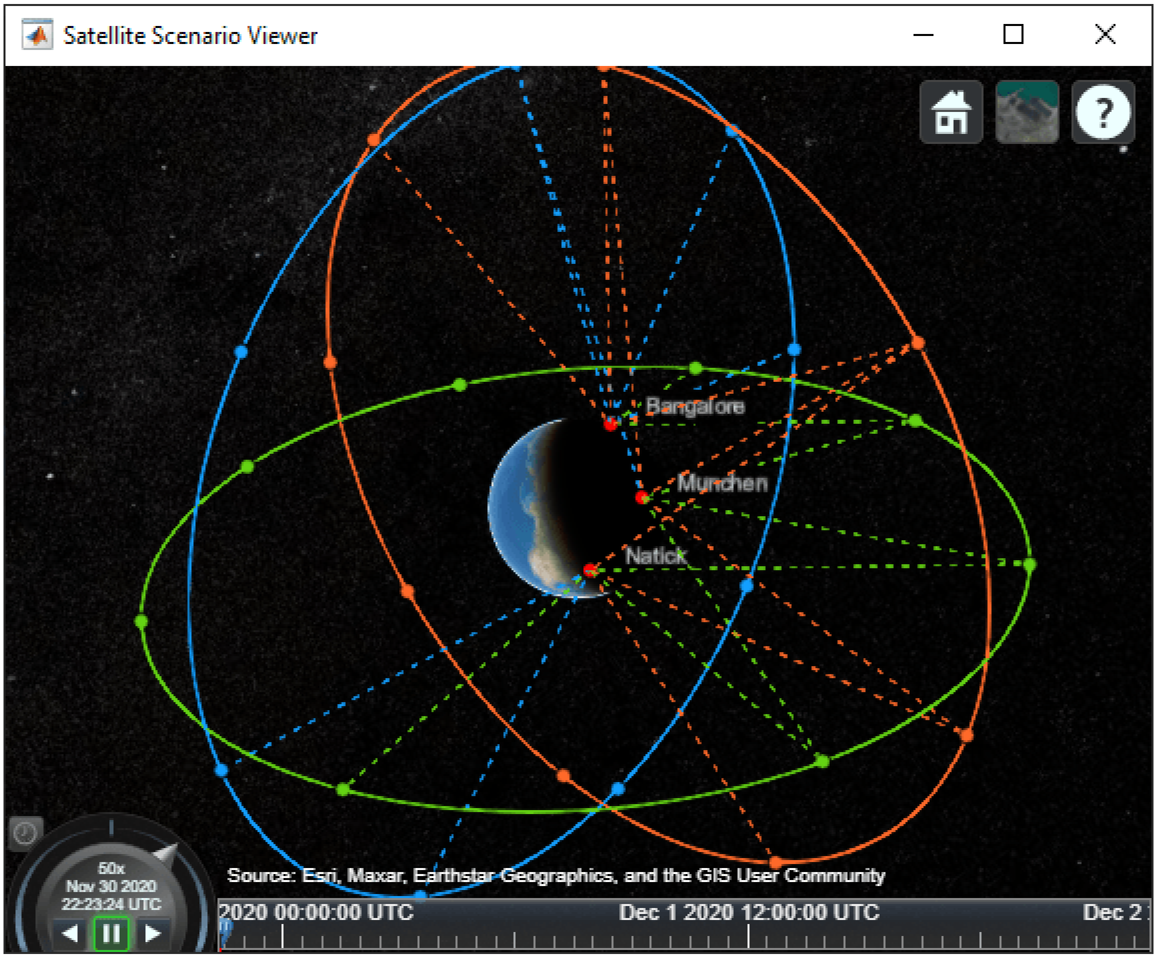
```
intervalsIN = accessIntervals(accessIN);
intervalsIN = sortrows(intervalsIN, "StartTime", "ascend")
```

```
intervalsIN=31×8 table
      Source          Target       IntervalNumber        StartTime                EndTime
    ___________    ____________    ______________    ____________________    ____________________

    "Bangalore"    "GALILEO 3"           1           30-Nov-2020 22:23:24    01-Dec-2020 05:12:2
    "Bangalore"    "GALILEO 4"           1           30-Nov-2020 22:23:24    01-Dec-2020 02:59:2
    "Bangalore"    "GALILEO 5"           1           30-Nov-2020 22:23:24    01-Dec-2020 00:22:2
    "Bangalore"    "GALILEO 9"           1           30-Nov-2020 22:23:24    01-Dec-2020 03:37:2
    "Bangalore"    "GALILEO 10"          1           30-Nov-2020 22:23:24    01-Dec-2020 00:09:2
    "Bangalore"    "GALILEO 16"          1           30-Nov-2020 22:23:24    01-Dec-2020 08:44:2
    "Bangalore"    "GALILEO 21"          1           30-Nov-2020 22:23:24    30-Nov-2020 23:25:2
    "Bangalore"    "GALILEO 22"          1           30-Nov-2020 22:23:24    30-Nov-2020 22:58:2
    "Bangalore"    "GALILEO 15"          1           01-Dec-2020 00:17:24    01-Dec-2020 11:16:2
    "Bangalore"    "GALILEO 2"           1           01-Dec-2020 00:25:24    01-Dec-2020 07:10:2
    "Bangalore"    "GALILEO 22"          2           01-Dec-2020 00:48:24    01-Dec-2020 05:50:2
    "Bangalore"    "GALILEO 21"          2           01-Dec-2020 01:32:24    01-Dec-2020 08:29:2
    "Bangalore"    "GALILEO 1"           1           01-Dec-2020 03:06:24    01-Dec-2020 07:17:2
    "Bangalore"    "GALILEO 20"          1           01-Dec-2020 03:36:24    01-Dec-2020 12:38:2
    "Bangalore"    "GALILEO 14"          1           01-Dec-2020 05:48:24    01-Dec-2020 13:29:2
    "Bangalore"    "GALILEO 19"          1           01-Dec-2020 05:53:24    01-Dec-2020 17:06:2
      ⋮
```

View the Satellite Scenario

Open a 3-D viewer window of the scenario. The viewer window contains all 24 satellites and the three ground stations defined earlier in this example. A line is drawn between each ground station and satellite during their corresponding access intervals. Hide the details of the satellites and ground stations by setting the `ShowDetails` name-value pair to `false`. Show satellite orbits and labels for the ground station locations.

```
viewer3D = satelliteScenarioViewer(scenario, ShowDetails=false);
show(sat.Orbit);
gsUS.ShowLabel = true;
gsUS.LabelFontSize = 11;
gsDE.ShowLabel = true;
gsDE.LabelFontSize = 11;
gsIN.ShowLabel = true;
gsIN.LabelFontSize = 11;
```

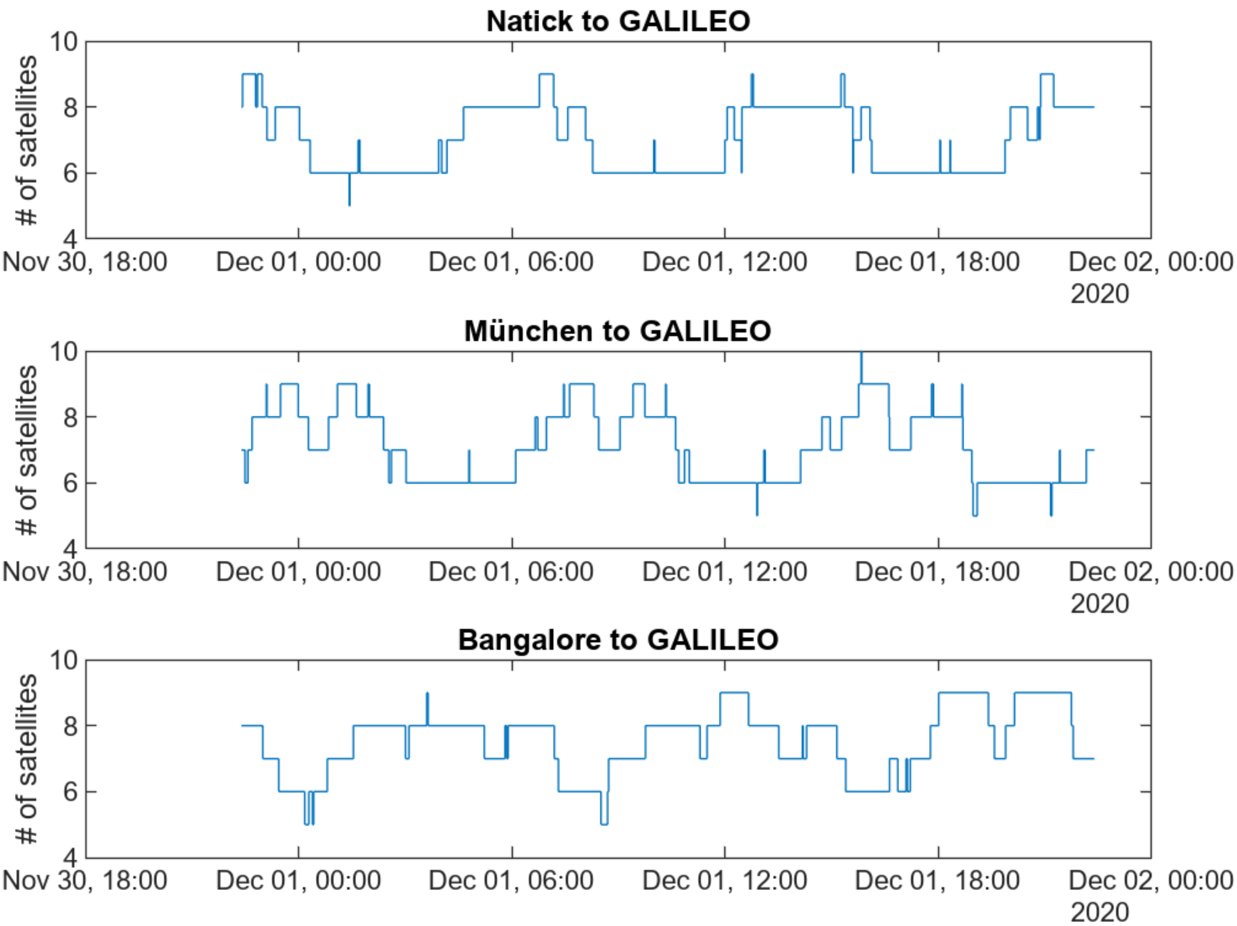


Compare Access Between Ground Stations

Calculate access status between each satellite and ground station using the accessStatus method. Each row of the output array corresponds with a satellite in the constellation. Each column corresponds with time steps in the scenario. A value of `True` indicates that the satellite can access the aircraft at that specific time sample. The second output of `accessStatus` contains the time steps of the scenario. Plot cumulative access for each ground station over the one day analysis window.

```
[statusUS, timeSteps] = accessStatus(accessUS);
statusDE = accessStatus(accessDE);
statusIN = accessStatus(accessIN);

% Sum cumulative access at each timestep
statusUS = sum(statusUS, 1);
statusDE = sum(statusDE, 1);
statusIN = sum(statusIN, 1);

subplot(3,1,1);
stairs(timeSteps, statusUS);
title("Natick to GALILEO")
ylabel("# of satellites")
subplot(3,1,2);
stairs(timeSteps, statusDE);
```

```
title("München to GALILEO")
ylabel("# of satellites")
subplot(3,1,3);
stairs(timeSteps, statusIN);
title("Bangalore to GALILEO")
ylabel("# of satellites")
```



Collect access interval metrics for each ground station in a table for comparison.

```
statusTable = [table(height(intervalsUS), height(intervalsDE), height(intervalsIN)); ...
    table(sum(intervalsUS.Duration)/3600, sum(intervalsDE.Duration)/3600, sum(intervalsIN.Duratio
    table(mean(intervalsUS.Duration/60), mean(intervalsDE.Duration/60), mean(intervalsIN.Duration
    table(mean(statusUS, 2), mean(statusDE, 2), mean(statusIN, 2)); ...
    table(min(statusUS), min(statusDE), min(statusIN)); ...
    table(max(statusUS), max(statusDE), max(statusIN))];
statusTable.Properties.VariableNames = ["Natick", "München", "Bangalore"];
statusTable.Properties.RowNames = ["Total # of intervals", "Total interval time (hrs)",...
    "Mean interval length (min)", "Mean # of satellites in view", ...
    "Min # of satellites in view", "Max # of satellites in view"];
statusTable
```

```
statusTable=6×3 table
                                   Natick    München    Bangalore
                                   ______    _______    _________

    Total # of intervals              40        40           31
    Total interval time (hrs)     167.88    169.95       180.42
    Mean interval length (min)    251.82    254.93       349.19
    Mean # of satellites in view   7.018    7.1041       7.5337
    Min # of satellites in view        5         5            5
    Max # of satellites in view        9        10            9
```

Walker-Delta constellations are evenly distributed across longitudes. Natick and München are located at similar latitudes, and therefore have very similar access characteristics with respect to the constellation. Bangalore is at a latitude closer to the equator, and despite having a lower number of individual access intervals, it has the highest average number of satellites in view, the highest overall interval time, and the longest average interval duration (by about 95 minutes). All locations always have at least 4 satellites in view, as is required for GNSS trilateration.

References

[1] Wertz, James R, David F. Everett, and Jeffery J. Puschell. *Space Mission Engineering: The New Smad*. Hawthorne, CA: Microcosm Press, 2011. Print.

[2] Beech, Theresa W., Sefania Cornana, Miguel B. Mora. *A Study of Three Satellite Constellation Design Algorithms.* 14th International Symposium on Space Flight Dynamics, Foz do Iguaçu, Brazil 1999.

[3] The European Space Agency: Galileo Facts and Figures. https://www.esa.int/Applications/Navigation/Galileo/Facts_and_figures

See Also

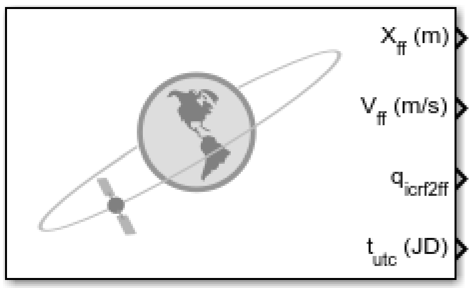
Blocks
Orbit Propagator

Objects
`satelliteScenario`

Mission Analysis with the Orbit Propagator Block

This example shows how to compute and visualize line-of-sight access intervals between satellite(s) and a ground station. It uses:

- Aerospace Blockset™ **`Orbit Propagator`** block
- Aerospace Toolbox **`satelliteScenario`** object
- Mapping Toolbox™ **`worldmap`** and **`geoshow`** functions



Orbit Propagator

The Aerospace Toolbox **`satelliteScenario`** object allows users to add satellites and constellations to scenarios in two ways. First, satellite initial conditions can be defined from a two line element file (.tle) or from Keplerian orbital elements and the satellites can then be propagated using Kepler's problem, simplified general perturbation algorithm SGP-4, or simplified deep space perturbation algorithm SDP-4. Additionally, previously generated timestamped ephemeris data can be added to a scenario from a timeseries or timetable object. Data is interpolated in the scenario object to align with the scenario time steps. This second option can be used to incorporate data generated in a Simulink® model into either a new or existing satelliteScenario. This example shows how to propagate satellite trajectories using numerical integration with the Aerospace Blockset **`Orbit Propagator`** block, and load that logged ephemeris data into a **`satelliteScenario`** object for access analysis.

Define Mission Parameters and Satellite Initial Conditions

Specify a start date and duration for the mission. This example uses MATLAB® structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

```
mission.StartDate = datetime(2019, 1, 4, 12, 0, 0);
mission.Duration  = hours(6);
```

Specify Keplerian orbital elements for the satellite(s) at the `mission.StartDate`.

```
mission.Satellite.SemiMajorAxis  = 6786233.13; % meters
mission.Satellite.Eccentricity   = 0.0010537;
mission.Satellite.Inclination    = 51.7519;    % deg
mission.Satellite.RAAN           = 95.2562;    % deg
mission.Satellite.ArgOfPeriapsis = 93.4872;    % deg
mission.Satellite.TrueAnomaly    = 202.9234;   % deg
```

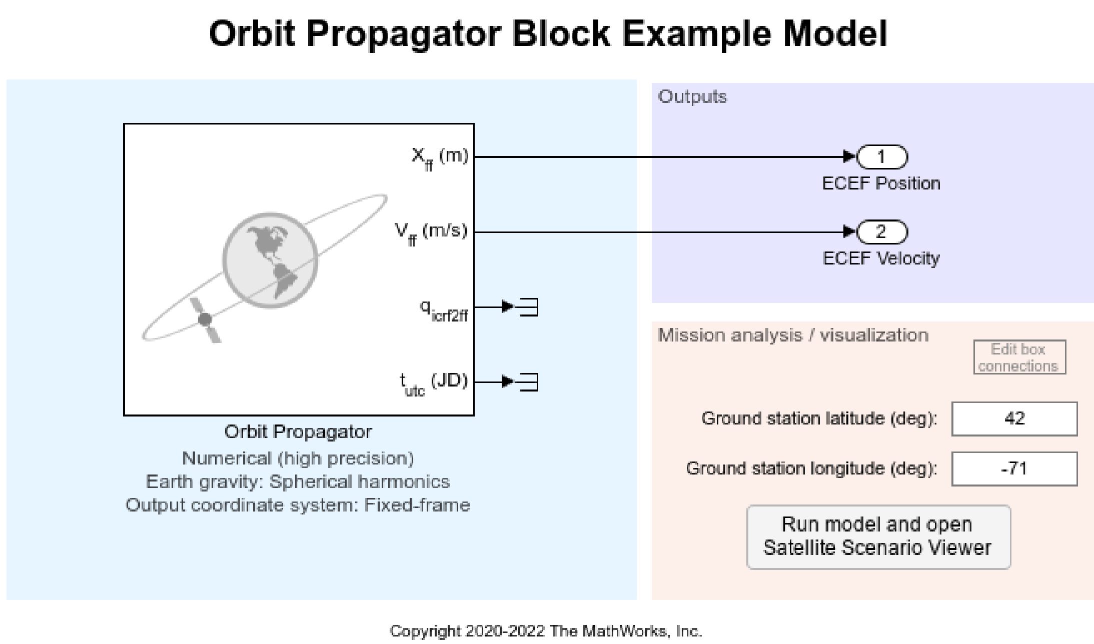
Specify the latitude and longitude of a ground station to use in access analysis below.

```
mission.GroundStation.Latitude  =  42 ; % deg
mission.GroundStation.Longitude = -71 ; % deg
```

Open and Configure the Orbit Propagation Model

Open the included Simulink model. This model contains an **`Orbit Propagator`** block connected to output ports. The **`Orbit Propagator`** block supports vectorization. This allows you to model multiple satellites in a single block by specifying arrays of initial conditions in the `Block Parameters` window or using `set_param`. The model also includes a "Mission Analysis and Visualization" section that contains a dashboard **`Callback button`**. When clicked, this button runs the model, creates a new satelliteScenario object in the global base workspace containing the satellite or constellation defined in the **`Orbit Propagator`** block, and opens a Satellite Scenario Viewer window for the new scenario. To view the source code for this action, double click the callback button. **The "Mission Analysis and Visualization" section is a standalone workflow to create a new satelliteScenario object and is not used as part of this example.**

```
mission.mdl = "OrbitPropagatorBlockExampleModel";
open_system(mission.mdl);
snapshotModel(mission.mdl);
```



Define the path to the **`Orbit Propagator`** block in the model.

```
mission.Satellite.blk = mission.mdl + "/Orbit Propagator";
```

Set satellite initial conditions. To assign the Keplerian orbital element set defined in the previous section, use `set_param`.

```
set_param(mission.Satellite.blk, ...
    "startDate",      num2str(juliandate(mission.StartDate)), ...
    "stateFormatNum", "Orbital elements", ...
    "orbitType",      "Keplerian", ...
    "semiMajorAxis",  "mission.Satellite.SemiMajorAxis", ...
    "eccentricity",   "mission.Satellite.Eccentricity", ...
    "inclination",    "mission.Satellite.Inclination", ...
    "raan",           "mission.Satellite.RAAN", ...
    "argPeriapsis",   "mission.Satellite.ArgOfPeriapsis", ...
    "trueAnomaly",    "mission.Satellite.TrueAnomaly");
```

Set the position and velocity output ports of the block to use the Earth-centered Earth-fixed frame, which is the International Terrestrial Reference Frame (ITRF).

```
set_param(mission.Satellite.blk, ...
    "centralBody",  "Earth", ...
    "outportFrame", "Fixed-frame");
```

Configure the propagator. This example uses a numerical propagator for higher position accuracy. Use numerical propagators to model Earth gravitational potential using the equation for universal gravitation (`"Pt-mass"`), a second order zonal harmonic model (`"Oblate Ellipsoid (J2)"`), or a spherical harmonic model (`"Spherical Harmonics"`). Spherical harmonics are the most accurate, but trade accuracy for speed. For increased accuracy, you can also specify whether to use Earth orientation parameters (EOP's) in the internal transformations between inertial (ICRF) and fixed (ITRF) coordinate systems.

```
set_param(mission.Satellite.blk, ...
    "propagator",   "Numerical (high precision)", ...
    "gravityModel", "Spherical Harmonics", ...
    "earthSH",      "EGM2008", ... % Earth spherical harmonic potential model
    "shDegree",     "120", ... % Spherical harmonic model degree and order
    "useEOPs",      "on", ... % Use EOP's in ECI to ECEF transformations
    "eopFile",      "aeroiersdata.mat"); % EOP data file
```

Apply model-level solver setting using `set_param`. For best performance and accuracy when using a numerical propagator, use a variable-step solver.

```
set_param(mission.mdl, ...
    "SolverType", "Variable-step", ...
    "SolverName", "VariableStepAuto", ...
    "RelTol",     "1e-6", ...
    "AbsTol",     "1e-7", ...
    "StopTime",   string(seconds(mission.Duration)));
```

Save model output port data as a dataset of time series objects.

```
set_param(mission.mdl, ...
    "SaveOutput", "on", ...
    "OutputSaveName", "yout", ...
    "SaveFormat", "Dataset");
```

Run the Model and Collect Satellite Ephemerides

Simulate the model. In this example, the **`Orbit Propagator`** block is set to output position and velocity states in the ECEF (ITRF) coordinate frame.

```
mission.SimOutput = sim(mission.mdl);
```

Extract position and velocity data from the model output data structure.

```
mission.Satellite.TimeseriesPosECEF = mission.SimOutput.yout{1}.Values;
mission.Satellite.TimeseriesVelECEF = mission.SimOutput.yout{2}.Values;
```

Set the start data from the mission in the timeseries object.

```
mission.Satellite.TimeseriesPosECEF.TimeInfo.StartDate = mission.StartDate;
mission.Satellite.TimeseriesVelECEF.TimeInfo.StartDate = mission.StartDate;
```

Load the Satellite Ephemerides into a satelliteScenario Object

Create a satellite scenario object to use during the analysis portion of this example.

```
scenario = satelliteScenario;
```

Add the satellites to the satellite scenario as ECEF position and velocity timeseries using the `satellite` method.

```
sat = satellite(scenario, mission.Satellite.TimeseriesPosECEF, mission.Satellite.TimeseriesVelECI
    "CoordinateFrame", "ecef")
```

```
sat = 
  Satellite with properties:

                 Name:  Satellite
                   ID:  1
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
              Eclipse:  [1x0 Aero.satellitescenario.Eclipse]
          GroundTrack:  [1x1 matlabshared.satellitescenario.GroundTrack]
                Orbit:  [1x1 matlabshared.satellitescenario.Orbit]
       CoordinateAxes:  [1x1 matlabshared.satellitescenario.CoordinateAxes]
      OrbitPropagator:  ephemeris
          MarkerColor:  [0.059 1 1]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
        Visual3DModel:  
   Visual3DModelScale:  1
```

```
disp(scenario)
```

```
  satelliteScenario with properties:

         StartTime: 04-Jan-2019 12:00:00
          StopTime: 04-Jan-2019 18:00:00
        SampleTime: 60
      AutoSimulate: 1
        Satellites: [1×1 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
         Platforms: [1×0 matlabshared.satellitescenario.Platform]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Preview latitude (deg), longitude (deg), and altitude (m) for each satellite. Use the `states` method to query satellite states at each scenario time step.

```
for idx = numel(sat):-1:1
    % Retrieve states in geographic coordinates
    [llaData, ~, llaTimeStamps] = states(sat(idx), "CoordinateFrame","geographic");
    % Organize state data for each satellite in a separate timetable
    mission.Satellite.LLATable{idx} = timetable(llaTimeStamps', llaData(1,:)', llaData(2,:)', lla
        'VariableNames', {'Lat_deg','Lon_deg', 'Alt_m'});
    mission.Satellite.LLATable{idx}
end
```

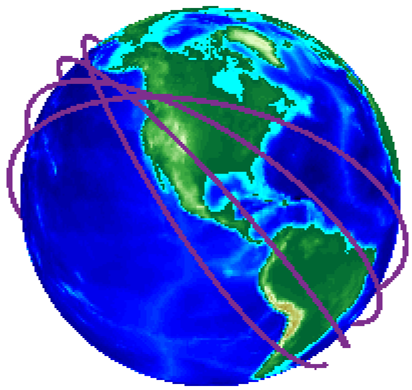
```
ans=361×3 timetable
            Time             Lat_deg    Lon_deg      Alt_m
    ____________________    _______    _______    __________

    04-Jan-2019 12:00:00    -44.804    120.35     4.2526e+05
    04-Jan-2019 12:01:00    -42.809     124.7     4.2232e+05
    04-Jan-2019 12:02:00    -40.638    128.75     4.2393e+05
    04-Jan-2019 12:03:00    -38.337     132.5     4.2008e+05
    04-Jan-2019 12:04:00    -35.867    136.05     4.2003e+05
    04-Jan-2019 12:05:00    -33.311    139.33     4.2031e+05
    04-Jan-2019 12:06:00    -30.682    142.38     4.1871e+05
    04-Jan-2019 12:07:00    -27.917    145.31     4.1982e+05
    04-Jan-2019 12:08:00    -25.104    148.06     4.1836e+05
    04-Jan-2019 12:09:00    -22.267    150.65     4.1404e+05
    04-Jan-2019 12:10:00    -19.321    153.17     4.1823e+05
    04-Jan-2019 12:11:00    -16.358    155.57     4.1717e+05
    04-Jan-2019 12:12:00    -13.397    157.88       4.07e+05
    04-Jan-2019 12:13:00     -10.36    160.15     4.1036e+05
    04-Jan-2019 12:14:00    -7.3121    162.37     4.1291e+05
    04-Jan-2019 12:15:00    -4.2727    164.54     4.0493e+05
      ⋮
```

```
clear llaData llaTimeStamps;
```

Display Satellite Trajectories over the 3D Globe

To display the satellite trajectories over Earth (WGS84 ellipsoid), use helper function `plot3DTrajectory`.

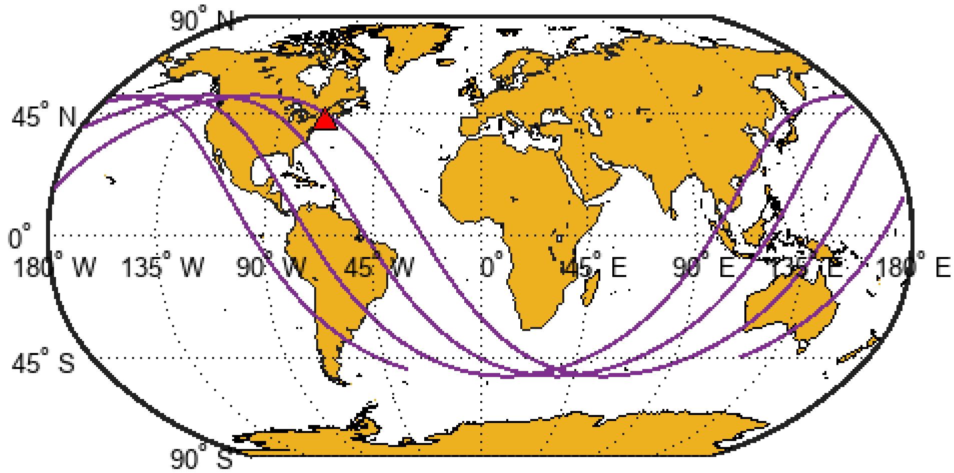
```
mission.ColorMap = lines(256); % Define colormap for satellite trajectories
mission.ColorMap(1:3,:) = [];
plot3DTrajectories(mission.Satellite, mission.ColorMap);
```



Display Global and Regional 2-D Ground Traces

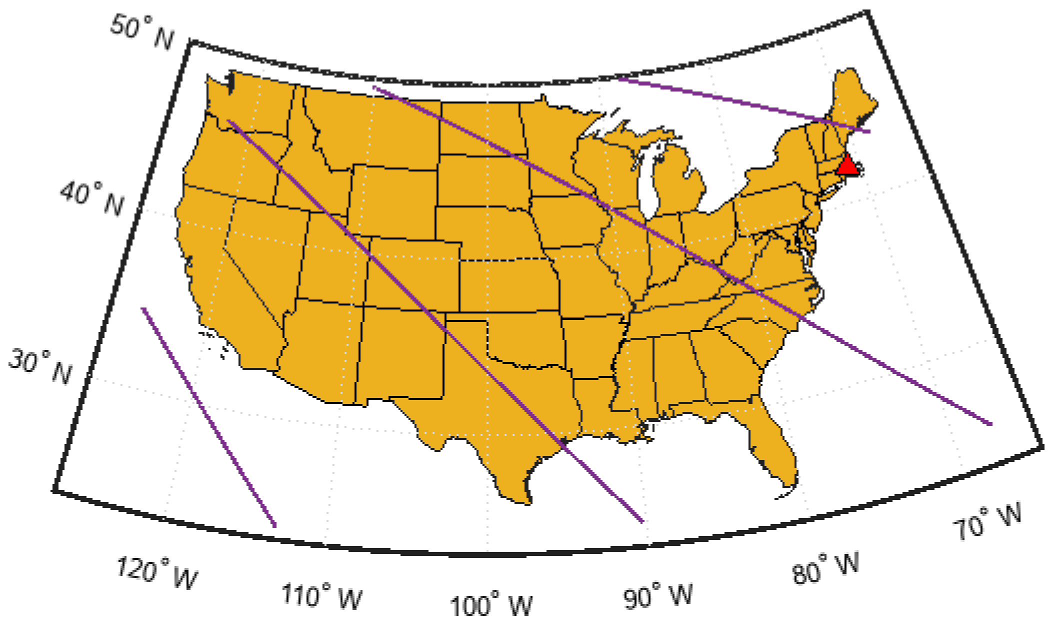
View the global ground trace as a 2-D projection using helper function `plot2DTrajectories`:

```
plot2DTrajectories(mission.Satellite, mission.GroundStation, mission.ColorMap);
```



View regional ground trace. Select the region of interest from the dropdown menu:

```
plot2DTrajectories(mission.Satellite, mission.GroundStation, mission.ColorMap,  Contiguous US
```



Compute Satellite to Ground Station Access (Line-of-Sight Visibility)

Add the ground station to the satelliteScenario object using the `groundStation` method.

```
gs = groundStation(scenario, mission.GroundStation.Latitude, mission.GroundStation.Longitude, ...
    "MinElevationAngle", 10, "Name", "Ground Station")
```

```
gs = 
  GroundStation with properties:

                 Name:  Ground Station
                   ID:  2
             Latitude:  42 degrees
            Longitude:  -71 degrees
             Altitude:  0 meters
    MinElevationAngle:  10 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
              Eclipse:  [1x0 Aero.satellitescenario.Eclipse]
       CoordinateAxes:  [1x1 matlabshared.satellitescenario.CoordinateAxes]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
```

```
        LabelFontSize:  15
```

Attach line-of-sight access analyses between all individual satellites and the ground station using the `access` method.

```
ac = access(sat, gs);
ac.LineColor = "green";
```

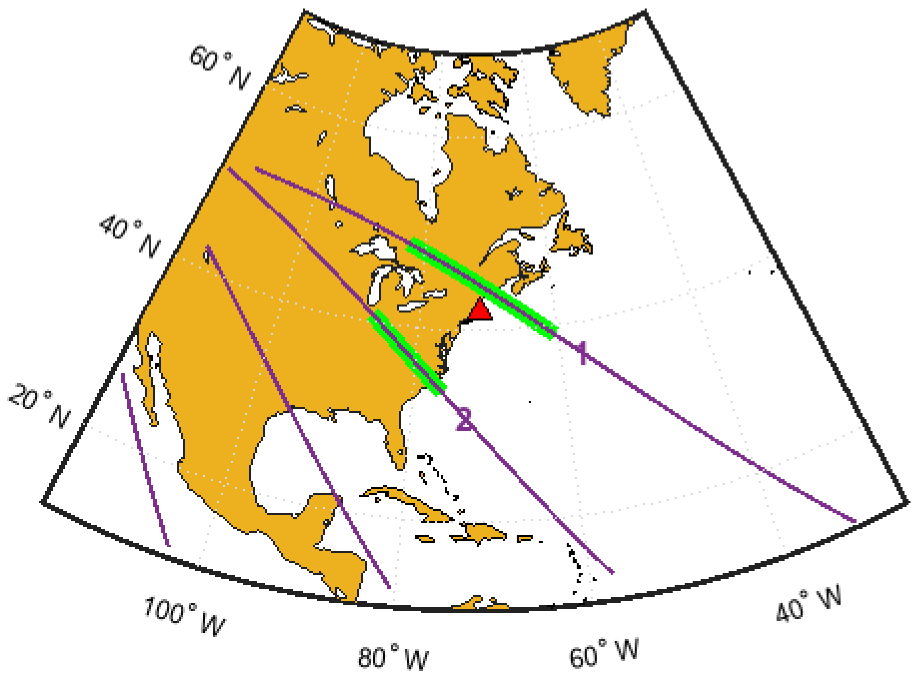
Display Access Intervals

Display access intervals for each satellite as a `timetable`. Use `accessStatus` and `accessIntervals` satellite methods to interact with access analysis results.

```
for idx = numel(ac):-1:1
    mission.Satellite.AccessStatus{idx} = accessStatus(ac(idx));
    mission.Satellite.AccessTable{idx} = accessIntervals(ac(idx));
    % Use local function addLLAToTimetable to add geographic positions and
    % closest approach range to the Access Intervals timetable
    mission.Satellite.AccessTable{idx} = addLLAToTimetable(...
        mission.Satellite.AccessTable{idx}, mission.Satellite.LLATable{idx}, mission.GroundStatio
end
clear idx;
```

Display access intervals overlaying 2-D ground traces of the satellite trajectories using helper function `plotAccessIntervals`.

```
plotAccessIntervals(mission.Satellite, mission.GroundStation, mission.ColorMap);
```



```
mission.Satellite.AccessTable{:}
```

```
ans=2×8 table
      Source           Target        IntervalNumber        StartTime               EndTime
    ___________    ________________    ______________    ____________________    ________________

    "Satellite"    "Ground Station"          1          04-Jan-2019 12:44:00    04-Jan-2019 12:5
    "Satellite"    "Ground Station"          2          04-Jan-2019 14:21:00    04-Jan-2019 14:2
```

Further Analysis

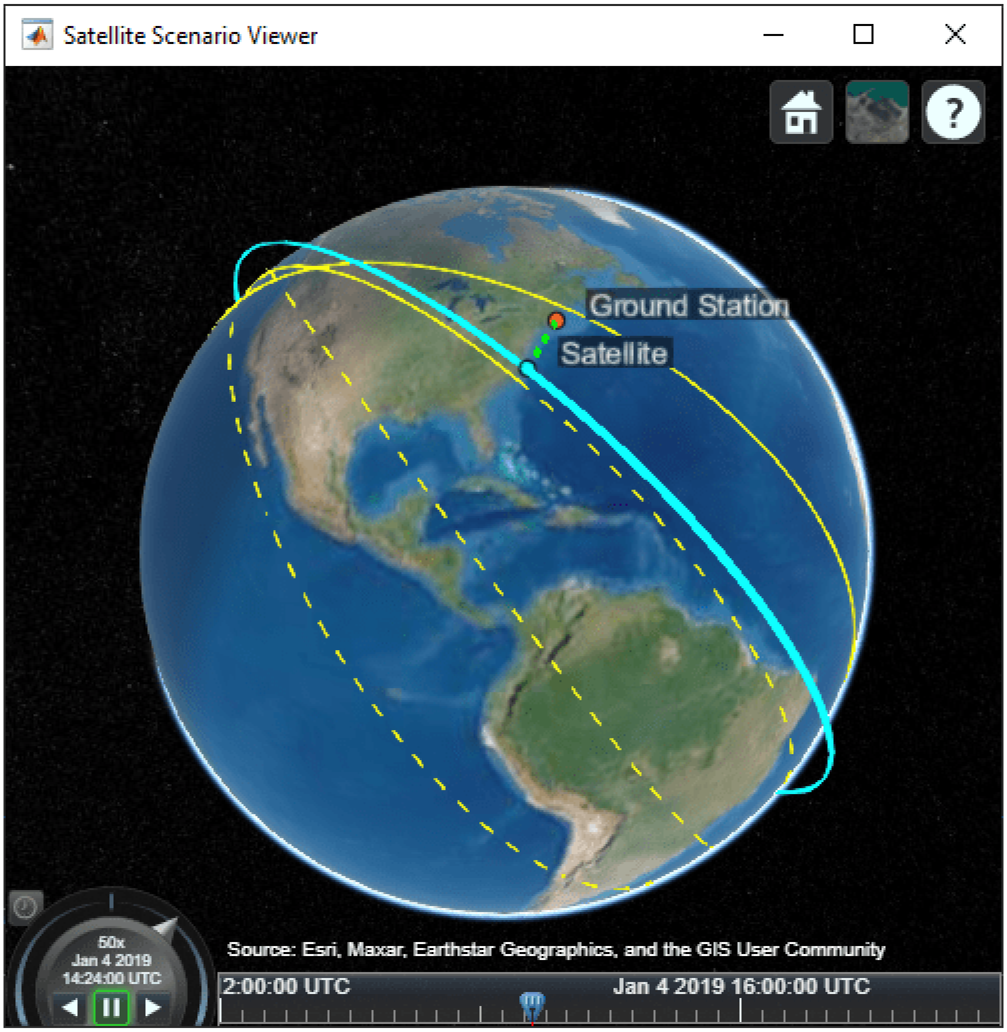
Play the `satelliteScenario` object to open and animate the scenario in a `satelliteScenarioViewer` window.

```
play(scenario);
disp(scenario.Viewers(1))
```

```
  Viewer with properties:

                      Name: 'Satellite Scenario Viewer'
                  Position: [560 240 800 600]
                   Basemap: 'satellite'
    PlaybackSpeedMultiplier: 50
       CameraReferenceFrame: 'ECEF'
                CurrentTime: 04-Jan-2019 12:02:26
                ShowDetails: 1
                  Dimension: '3D'
```

Show the satellite ground track in the viewer.

```
groundTrack(sat);
```



References

[1] Wertz, James R, David F. Everett, and Jeffery J. Puschell. *Space Mission Engineering: The New Smad*. Hawthorne, CA: Microcosm Press, 2011. Print.

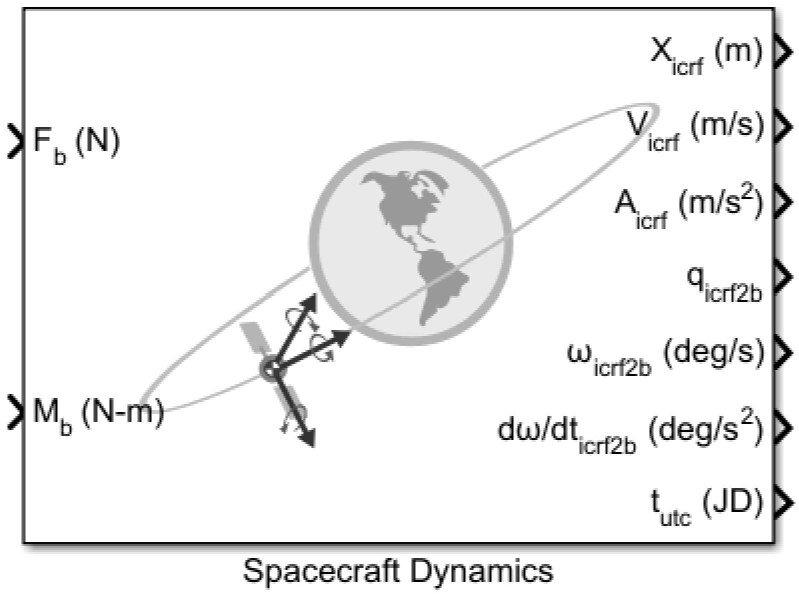
See Also

Blocks
Orbit Propagator

Objects
`satelliteScenario`

Getting Started with the Spacecraft Dynamics Block

This example shows how to model six degree-of-freedom rigid-body dynamics of a spacecraft or constellation of spacecraft with the Spacecraft Dynamics block from Aerospace Blockset™.



The Spacecraft Dynamics block models translational and rotational dynamics of spacecraft using numerical integration. It computes the position, velocity, attitude, and angular velocity of one or more spacecraft over time. For the most accurate results, use a variable step solver with low tolerance settings (less than 1e-8). Depending on your mission requirements, you can increase speed by using larger tolerances. Doing so may impact the accuracy of the solution.

Define orbital states as a set of orbital elements or as position and velocity state vectors. To propagate orbital states, the block uses the gravity model selected for the current central body. The block also includes external accelerations and forces provided as inputs to the block.

Attitude states are defined using quaternions, direction cosine matrices (DCMs), or Euler angles. To propagate attitude states, the block uses moments provided as inputs to the block and mass properties defined on the block.

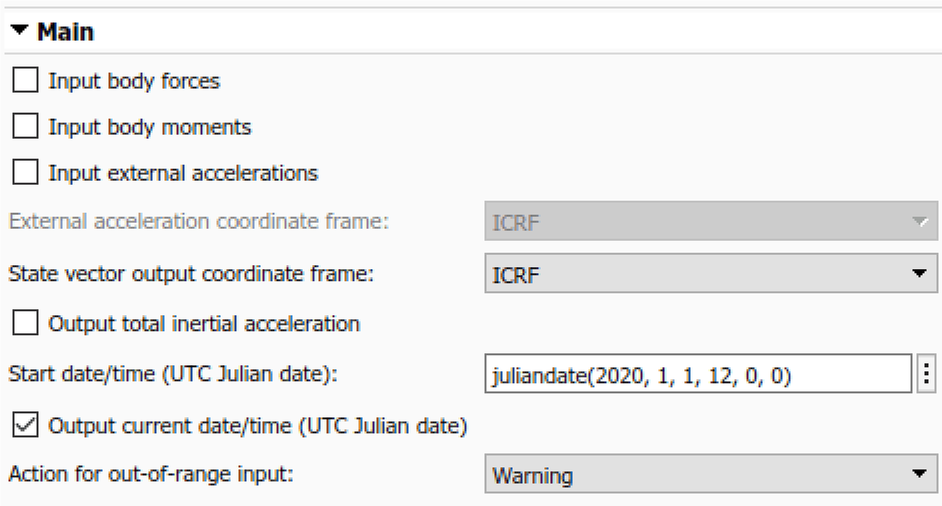
This document walks through the various options and configurations available on the block, explains how to model a constellation of spacecraft, and presents an example Simulink® model that implements the Spacecraft Dynamics block for a low-Earth observation satellite. Finally, the equations used by the block are presented.

Block Description

The Spacecraft Dynamics block can be found in the Simulink Library Browser (Aerospace Blockset→ Spacecraft→Spacecraft Dynamics), or by typing "Spacecraft Dynamics" into the quick insert dialog on the Simulink model canvas. This section provides an overview of the options available on the block, viewed from the Simulink Property Inspector (on the **Modeling** tab, under **Design**).

Main Tab

The **Main** tab includes block-level configuration parameters. All parameters on this tab apply to every spacecraft defined in the block.



You can specify whether to include:

- **Body forces** defined in the body frame
- **Body moments** defined in the Body frame ports
- **External accelerations,** to include perturbing accelerations in orbit propagation that are not included in the block's internal calculations. By default, the block calculates and uses central body gravity for orbit propagation (see **Central Body** tab on page 9-126 below). Some examples of additional perturbing accelerations that you can include in propagation are those due to atmospheric drag, third body gravity, and solar radiation pressure. You can provide perturbing accelerations in the inertial (ICRF) or fixed-frame coordinate systems, depending on the value set for **External acceleration coordinate frame**. For more information about fixed-frame coordinate systems used for each central body, see the Coordinate Systems section of the block reference page.

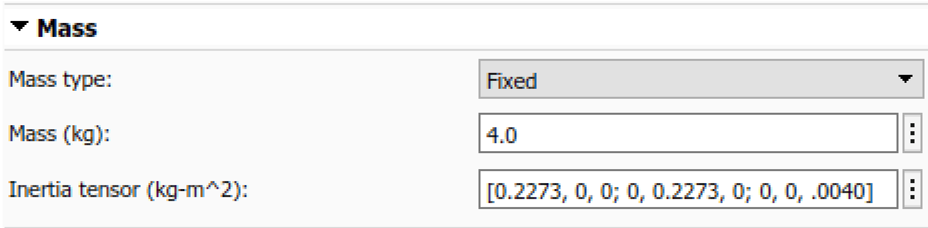
State vector output coordinate frame controls whether position and velocity state outputs from the block are in the inertial (ICRF) or fixed-frame coordinate systems.

You can also specify whether to output total inertial acceleration from the block, which is always in the inertial frame. This value is the total acceleration, including internally computed central body gravity as well as contributions from body forces and external accelerations provided to the block as inputs. Note, the acceleration output port is intended for diagnostic use only. It is not a valid workflow to feed this signal back into the block as an input.

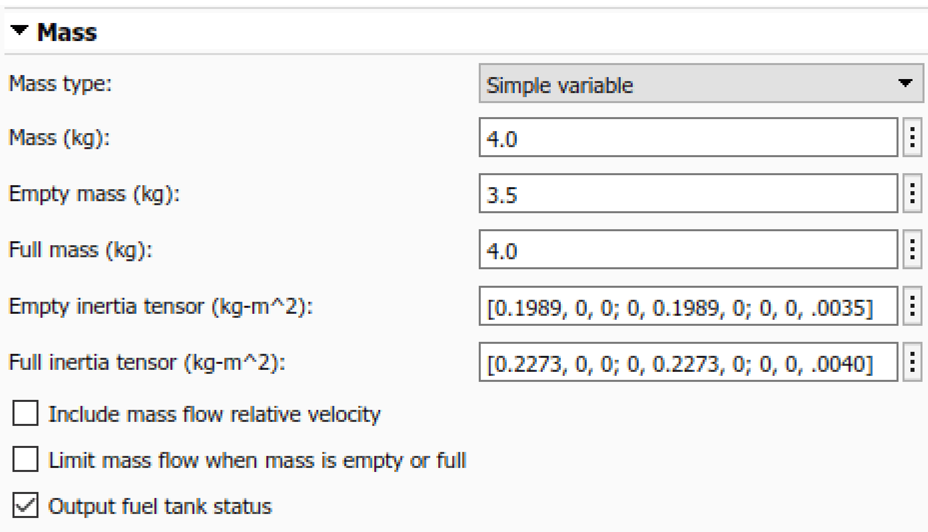
Start date/time is the initial date/time corresponding with the Simulink model start time t_0. It is the assumed epoch for all initial conditions provided on the block. Optionally, you can select **Output current date/time** to output a time signal from the block to use elsewhere in the simulation.

Mass Tab

Three mass types are available to model mass properties of the spacecraft: `Fixed`, `Simple variable`, and `Custom Variable`.



When **Mass type** is `Fixed`, the mass and inertia tensor are held constant at the values provided for **Mass** and **Inertia tensor** throughout the simulation. Mass flow rate and rate of change of inertia equal zero.



When **Mass type** is `Simple variable`, a simplistic approach is taken to vary the mass properties of the spacecraft during the simulation.

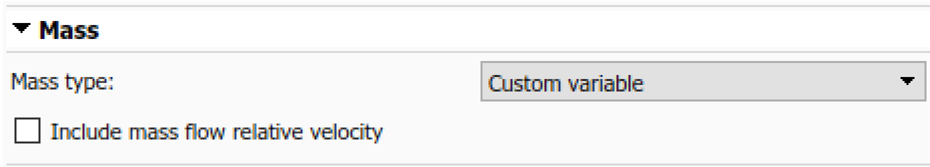
An initial **Mass**, **Empty mass** (dry), and **Full mass** (wet) are defined. Mass flow rate is provided to the block via an input port (**`dm/dt`**). This value is integrated to calculate the current mass at each time step of the simulation.

Similarly, in this configuration you provide inertia tensor values for the empty and full spacecraft configurations. The current tensor is approximated by linear interpolation between **Empty inertia tensor** and **Full inertia tensor** based on the current mass value.

You can optionally add another input port (V_{re_b}) to the block to provide a mass flow relative velocity using parameter **Include mass flow relative velocity**. This relative velocity is provided in the Body frame. It is used to calculate the force contribution due to mass being ablated from or added to the spacecraft.

To limit the mass flow rate provided to the block when the current mass is below the empty mass or above the full mass value, use parameter **Limit mass flow when mass is empty or full**.

Finally, the current fuel status can be output from the block (**`Fuel Status`**) based on the current mass. If the current mass exceeds **Full mass**, the reported status is 1. If the current mass is below **Empty mass**, the status is -1. When the mass is within the provided operating range, the status is 0.



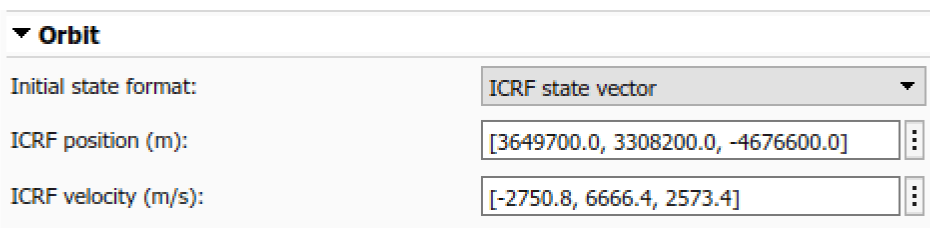
When **Mass type** is `Custom variable`, more flexibility is provided regarding how the mass properties of the spacecraft change over time. However this requires that more values be calculated externally from the block.

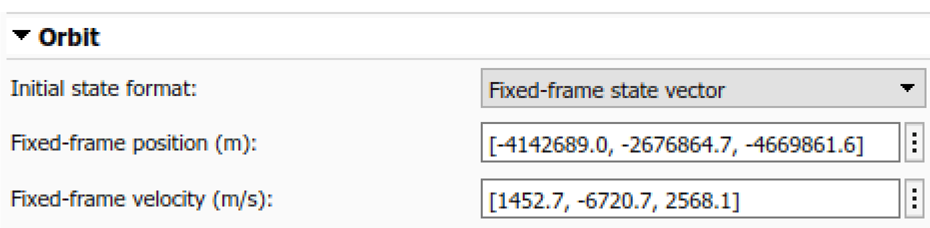
In this configuration, block input ports are added for the current mass (**m**), current inertia tensor (**I**), and the current rate of change of the inertia tensor (**dI/dt**).

To provide mass flow relative velocity, you can optionally add another input port (V_{re_b}) to the block with the **Include mass flow relative velocity** parameter. This relative velocity is provided in the Body frame. It is used to calculate the force contribution due to mass being ablated from or added to the spacecraft. Therefore, enabling this parameter adds an additional port to the block to provide mass flow rate (**dm/dt**).

Orbit Tab

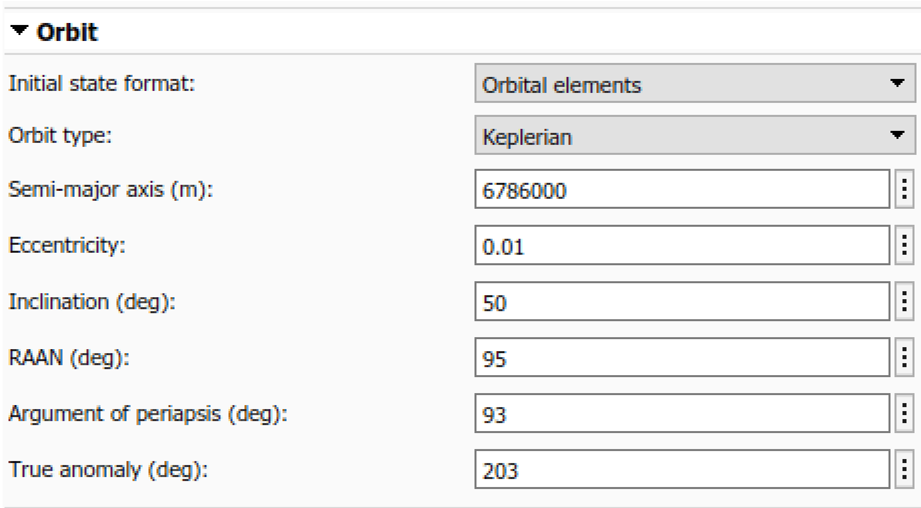
The **Orbit** tab defines initial conditions for the spacecraft as sets of orbital elements or as position and velocity state vectors depending on the value of **Initial state format** (`Orbital elements`, `ICRF state vector`, or `Fixed-frame state vector`).





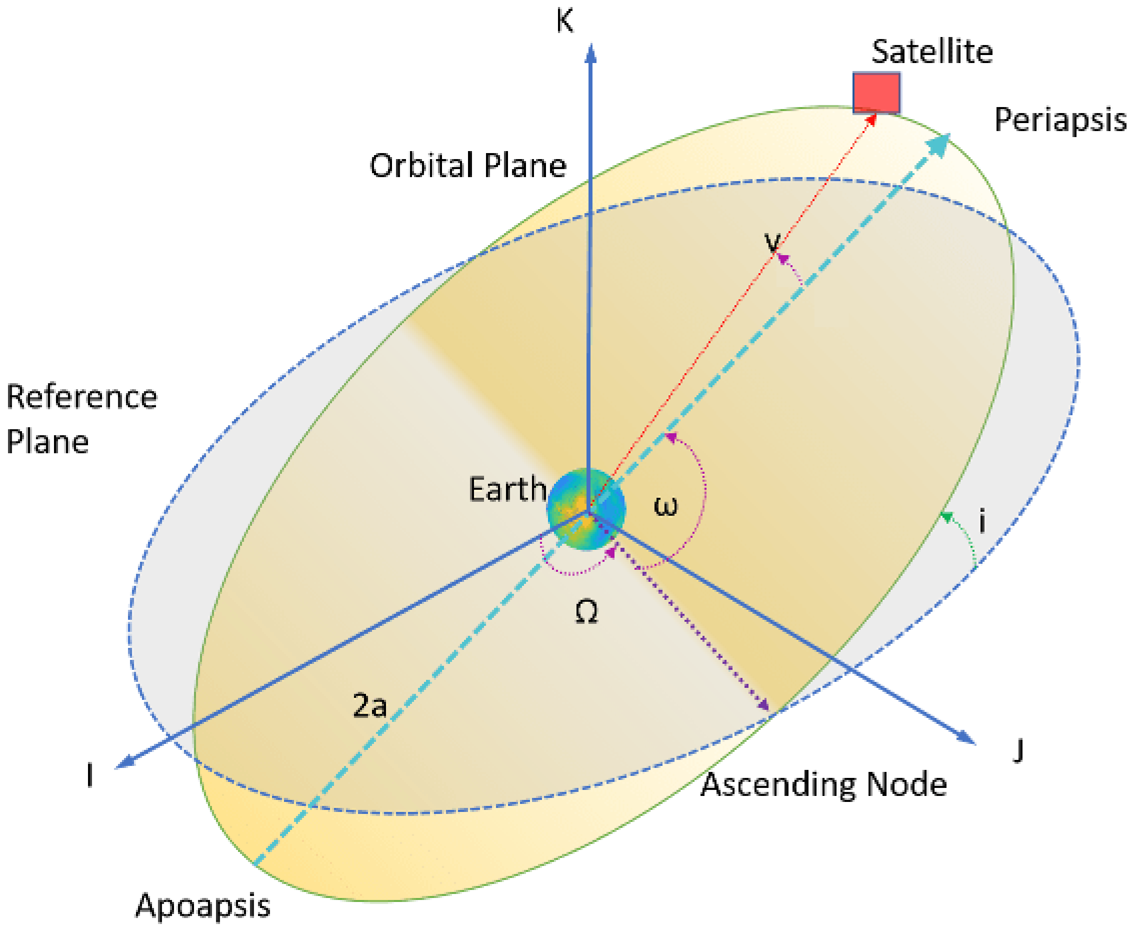
For state vector options, provide the initial position and velocity that correspond with **Start date/time** in the specified coordinate frame.

The **Initial state format** option `Orbital elements` is further decomposed by parameter **Orbit type**.



When **Orbit type** is `Keplerian`, you specify the traditional set of six Keplerian orbital elements:

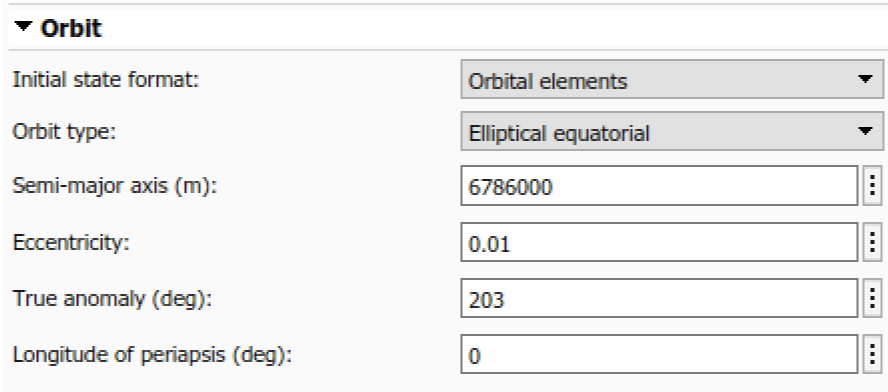
- **Semi-major axis** (a)
- **Eccentricity** (e)
- **Inclination** (i)
- Right ascension of the ascending node - **RAAN** (Ω)
- **Argument of periapsis** (ω)
- **True anomaly** (ν).



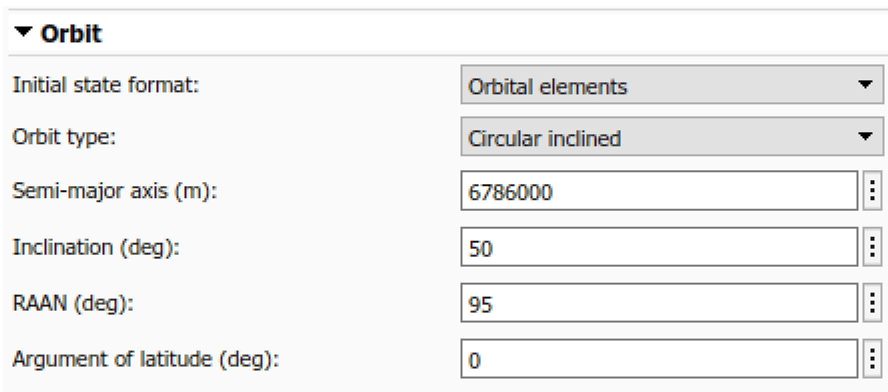
When specifying orbital elements, three orbit types result in undefined elements:

- When an orbit is equatorial (inclination equal to zero), RAAN is undefined.
- When an orbit is circular (eccentricity equal to zero), argument of periapsis and true anomaly are undefined.
- When an orbit is circular *and* equatorial, all three elements are undefined.

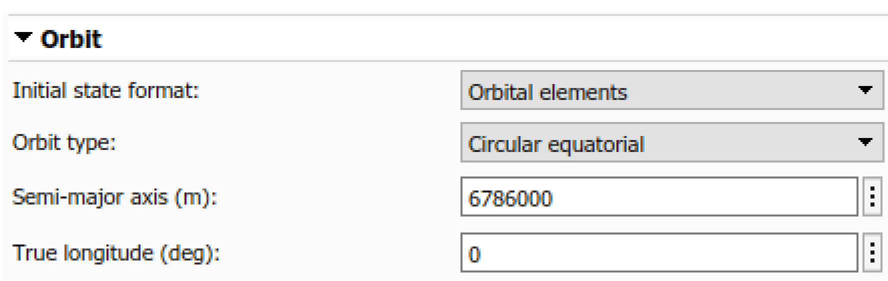
To assist in modeling these conditions, the block provides three additional options for **Orbit type** in addition to `Keplerian`.



For non-circular (elliptical) equatorial orbits, inclination always equals zero, and **RAAN** and **Argument of periapsis** are replaced with **Longitude of periapsis**. Longitude of periapsis is the angle between the ICRF X-axis (I) and periapsis. It is equal to the sum of RAAN (Ω) and the argument of periapsis (ω).



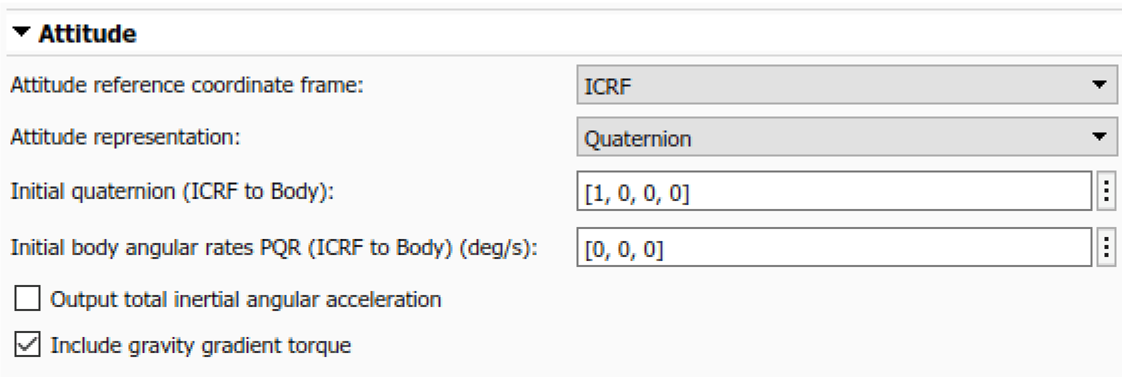
For circular inclined (non-equatorial) orbits, eccentricity always equals zero, and **Argument of periapsis** and **True anomaly** are replaced by **Argument of latitude**. Argument of latitude is the angle between the ascending node and the satellite position vector. It is equal to the sum of the true anomaly (ν) and the argument of periapsis (ω).



Finally, for circular equatorial orbits, inclination and eccentricity always equal zero, and **RAAN**, **Argument of periapsis**, and **True anomaly** are replaced by **True longitude**. True longitude is the angle between the ICRF X-axis (I) and the spacecraft position vector. It is equal to sum of true anomaly (ν), argument of periapsis (ω), and RAAN (Ω).

Attitude Tab

The **Attitude** tab defines initial conditions for the attitude of the spacecraft being modeled. Using parameter **Attitude reference coordinate frame**, you can define attitude with respect to the inertial (ICRF) frame, Fixed-frame, North-East-Down (NED) frame, or local-vertical local-horizontal (LVLH) frame. Initial attitude and body angular rate parameters provided to the block are assumed to be defined with respect to the specified frame. The attitude and body angular rate outputs from the block also use this frame.



To specify what representation method to use for attitude, use the **Attitude representation** parameter. Depending on the value selected, the **Initial attitude** parameter displays as **Initial quaternion**, **Initial DCM**, or **Initial Euler angles**. It expects the dimensions of the provided initial condition to match that representation. Additionally, the attitude output port from the block uses the specified representation.

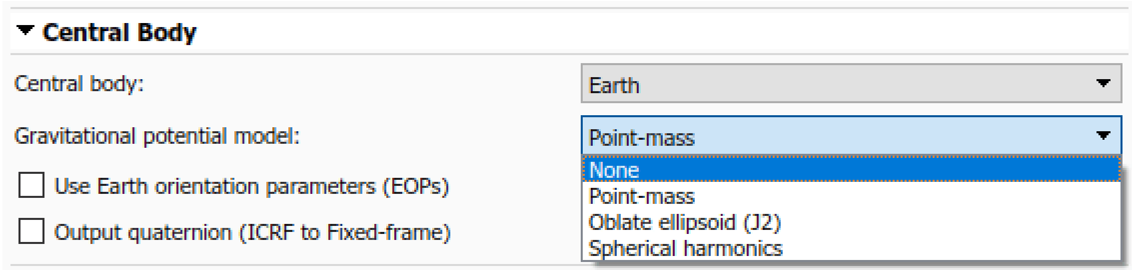
The initial attitude rate of change, **Initial body angular rates PQR,** and the corresponding output port (ω) are always defined as angular rates, regardless of the selection made for **Attitude representation**.

You can also specify whether to **Output total inertial angular acceleration** ($\dot{\omega}$) from the block. This output is always defined with respect to the inertial (ICRF) frame. If **Include gravity gradient torque** is selected, this value is the total angular acceleration due to moments provided as inputs to the block, and gravity gradient torque computed internally. Note, the angular acceleration output port is intended for diagnostic use only. It is not a valid workflow to feed this signal back into the block as an input.

If enabled, gravity gradient torque calculations treat the central body as a spherical body. The overall contribution due to gravity gradient torque is small. Treating the central body as a spherical body is generally sufficient for most applications. If a higher level of accuracy is required, gravity gradient torque values can be calculated externally and provided to the block as a moment. See the block equations section below for the equations implemented by the block.

Central Body Tab

Use the **Central Body** tab to provide information about the physical properties, gravitational potential model, and orientation of the celestial body around which the spacecraft is in orbit. All planets in our solar system are available, as well as the Earth Moon (Luna). Custom central bodies may also be defined. First, look at the various options for parameter **Gravitational potential model** (`None`, `Point-mass`, `Oblate ellipsoid (J2)`, and `Spherical harmonics`).



`None` is available for all central bodies. This option does not include any internally calculated gravitational acceleration in the system equations. Use this option in conjunction with the external acceleration input port if you have your own gravity model that you would like to use. When using option `None` with a custom central body, only planetary **Rotational rate** is required.

`Point-mass` is available for all central bodies. This option treats the central body as a point mass, and computes gravitational acceleration using Newtons law of universal gravitation. When using option `Point-mass` with a custom central body, you must provide **Equatorial radius**, **Flattening**, **Gravitational parameter** (μ), and planetary **Rotational rate**.

`Oblate ellipsoid (J2)` is available for all central bodies. This option includes the perturbing effects of the second-degree, zonal harmonic gravity coefficient, J_2, accounting for the oblateness of the central body. When using option `Oblate ellipsoid (J2)` with a custom central body, you must provide **Equatorial radius**, **Flattening**, **Gravitational parameter** (μ), **Second degree zonal harmonic (J2)**, and planetary **Rotational rate**.

`Spherical harmonics` is available only when central body is set to `Earth`, `Moon`, `Mars`, or `Custom`. The **Spherical harmonic model** options available for each central body are listed in this table:

Central body	Spherical Harmonic Model Option
Earth	EGM2008, EGM96, or EIGEN-GL04C
Moon	LP-100K or LP-165P
Mars	GMM2B

You must also specify a value for **Degree** that is below the maximum degree supported by the selected spherical harmonic model. Recommended and maximum degree values for each model are provided below:

Planet Model	Recommended Degree	Maximum Degree
`EGM2008`	120	2159
`EGM96`	70	360
`LP100K`	60	100
`LP165P`	60	165
`GMM2B`	60	80
`EIGENGL04C`	70	360

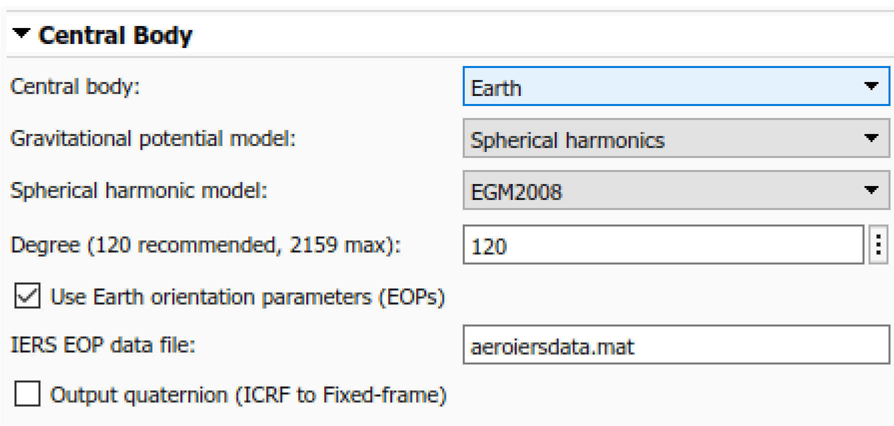
When using option `Spherical harmonics` with a custom central body, you must provide planetary **Rotational rate**, a **Spherical harmonic coefficient file** (.mat), and **Degree**. For more information about this file, see the parameter description in the Spacecraft Dynamics block reference page.

All planetary constants used by the block are from NASA JPL Planetary and Lunar Ephemerides DE405.

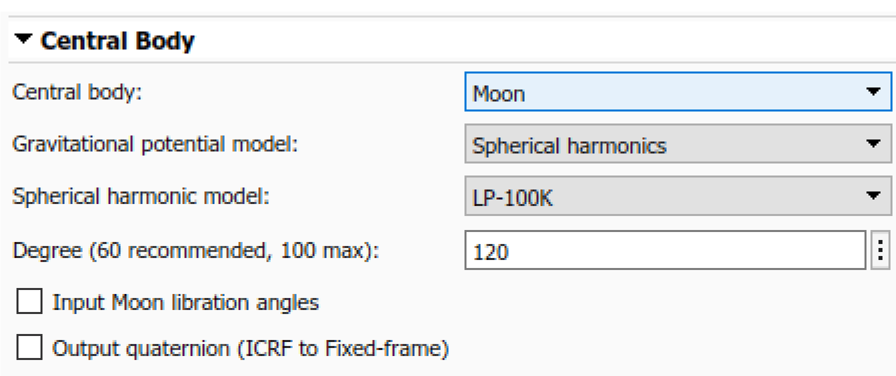
All J2 constant values are from NASA Space Science Data Coordinated Archive (NSSDCA).

If you need alternate constant values, use the `Custom` option for **Central body.**

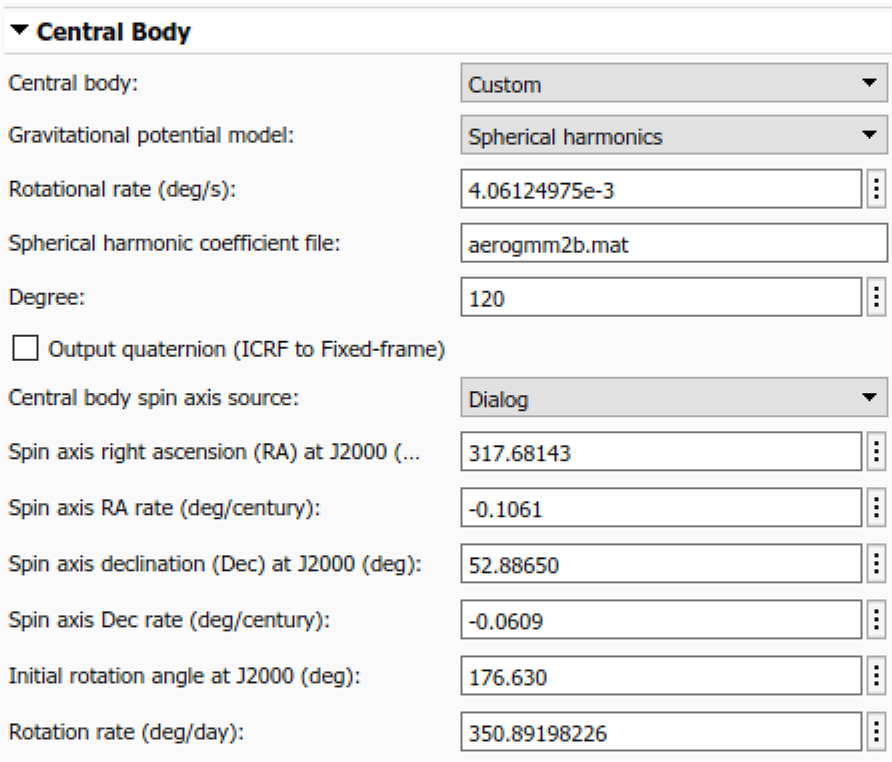
In addition to gravity, the **Central Body** tab includes information about the orientation of the central body. Available parameters depend on the currently selection **Central body.** All central bodies except `Earth`, `Moon`, and `Custom` use planetary rotational pole and meridian definitions from the *Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006*. Options specific to Earth, Moon, and Custom are discussed below.



When **Central body** is `Earth`, the fixed-frame coordinate system used by the block is the ITRF. By default, the transformation between ICRF and ITRF uses Earth orientation parameter (EOP) data provided to parameter **IERS EOP data file**. To generate an up-to-date EOP data file, use the Aerospace Toolbox function `aeroReadIERSData()`. This function calls out to the IERS data server and saves up-to-date EOP data to a MAT-file. To exclude Earth orientation parameter data from the transformation, clear **Use Earth orientation parameters (EOPs)**.



When **Central body** is `Moon`, you can provide Moon libration angles as inputs to the block by selecting **Input Moon libration angles**. When this option is selected, an input port is added to the block. In this case, libration angles are supplied at each time step of the simulation to use in the transformation between the fixed frame and ICRF. You can compute libration angles using the **`Moon Libration`** block. When using libration angles, the fixed-frame coordinate system for `Moon` is the Mean Earth/pole axis frame (ME). This frame is realized by two transformations. First, the block transforms values in the ICRF frame to the Principal Axis system (PA), which is the axis defined by the libration angles provided as inputs to the block. The block then transforms states into the ME system using a fixed rotation from *the Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006*. If the **Input Moon libration angles** check box is cleared, the fixed frame is defined by the directions of the poles of rotation and prime meridians defined in the *Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006*.

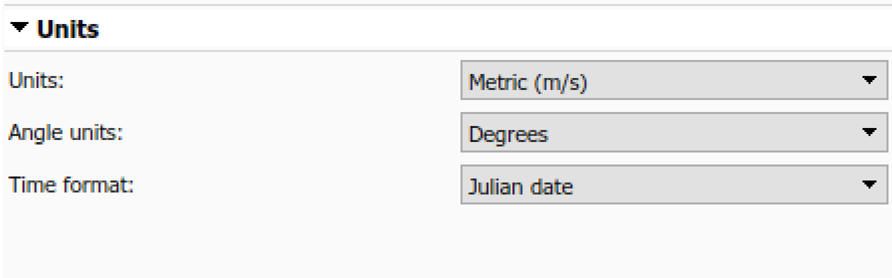


When **Central body** is `Custom`, there are two options to provide rotation pole and meridian data to the block, depending on the value of parameter **Central body spin axis source**. To provide current right ascension, declination, and rotational rate values as inputs to the block at each timestep, set the source to `Port`. To provide initial conditions for right ascension, declination, and rotation angle at J2000 (JD 2451545.0, i.e. 2000 January 1 12 hours TDB) as well as corresponding rate of change for each value, set the source to `Dialog`. These parameters align with the terminology and equations presented in the *Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006*.

Finally, for all central bodies, you can optionally output a quaternion that performs a position transformation from ICRF to the fixed-frame by selecting **Output quaternion (ICRF to Fixed-frame)**.

Units Tab

The **Units** tab defines the unit system, the **Angle units** (`Degrees` or `Radians`), and the time format used by the block.

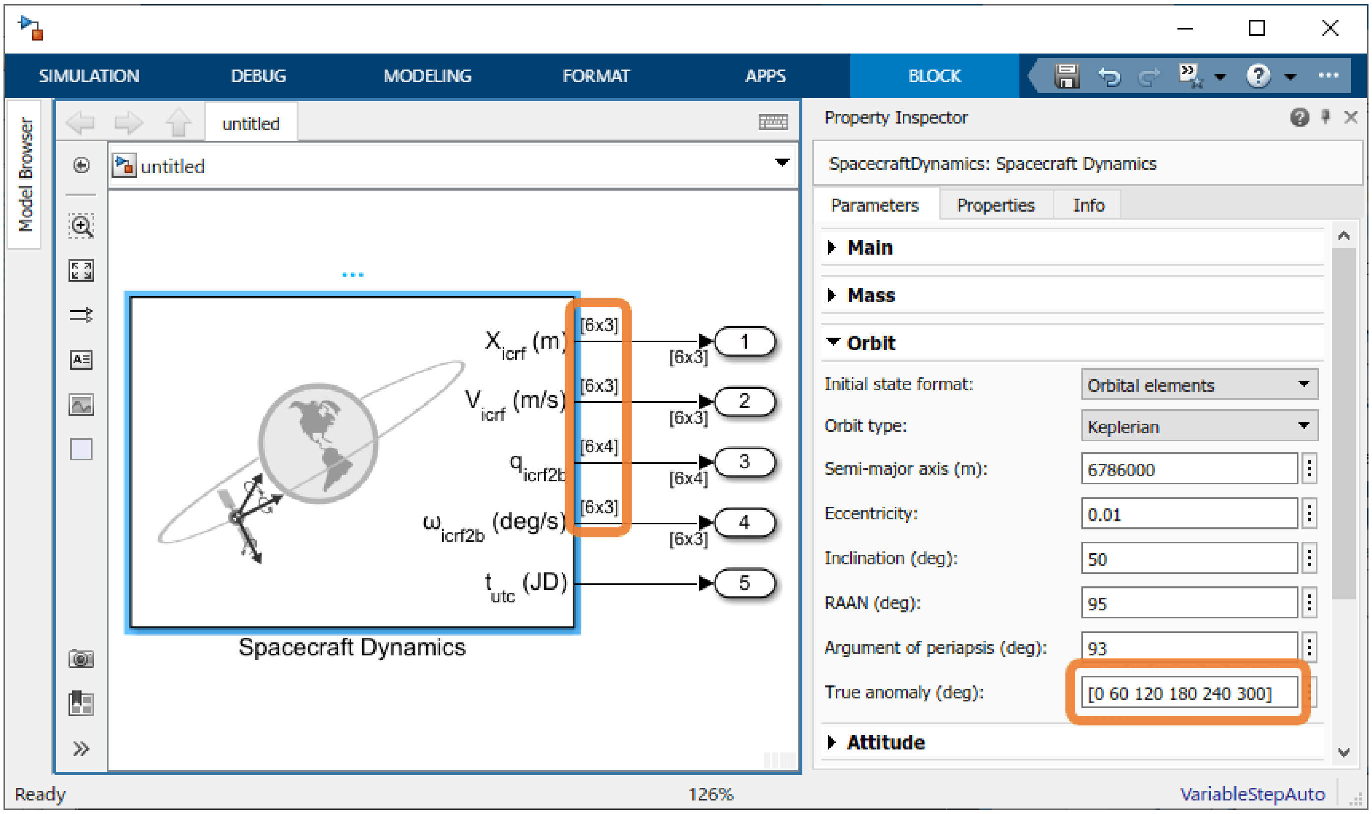


When **Time format** is `Julian date`, **Start date/time** and the block optional time output port use a scalar Julian date value. When set to `Gregorian`, both values are a 1x6 array of the form [Year, Month, Day, Hour, Minute, Second]. The corresponding units for each option of parameter **Units** are presented in the table below. Expected units in each parameter and port label on the block are updated automatically when **Units** is changed.

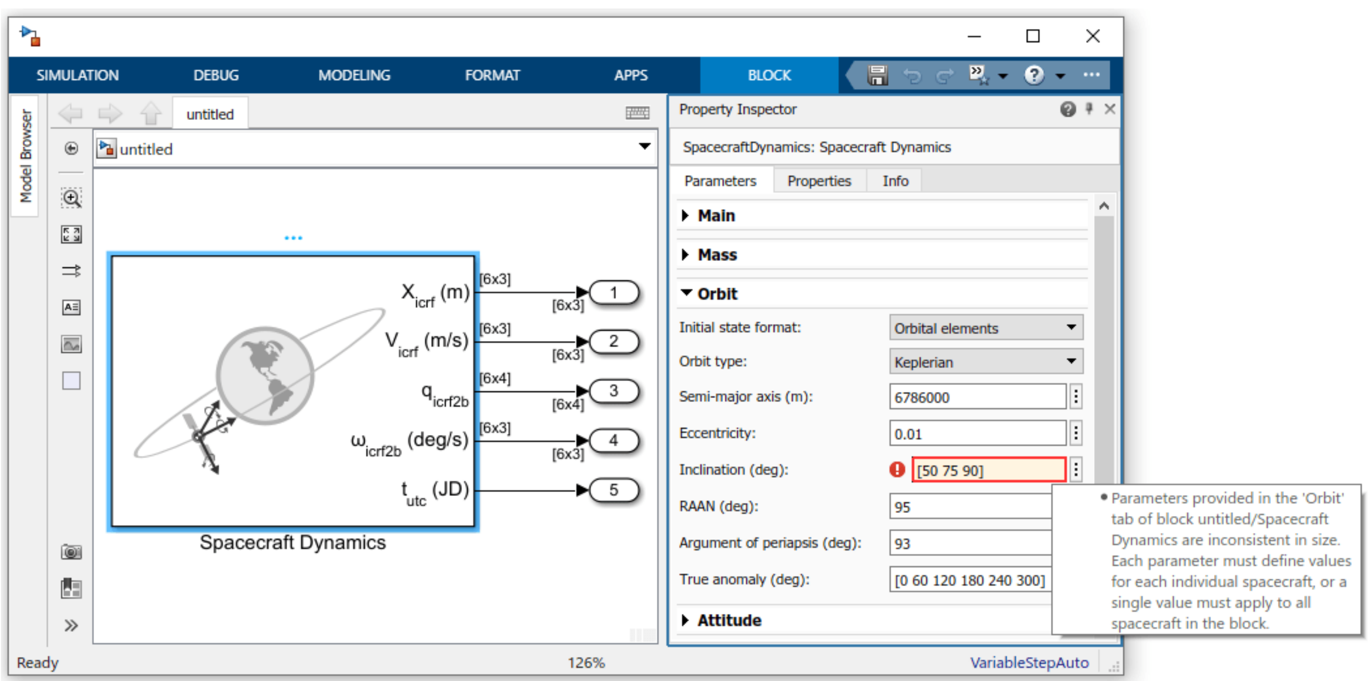
Units	Forces	Moment	Mass	Inertia	Distance Units	Velocity Units	Acceleration Units
`Metric (m/s)`	Newton	Newton meter	Kilograms	Kilogram m^2	meters	meters/sec	meters/sec^2
`Metric (km/s)`	Newton	Newton meter	Kilograms	Kilogram m^2	kilometers	kilometers/sec	kilometers/sec^2
`Metric (km/h)`	Newton	Newton meter	Kilograms	Kilogram m^2	kilometers	kilometers/hour	kilometers/$hour^2$
`English (ft/s)`	Pound-force	Foot-pound	Slugs	Slug ft^2	feet	feet/sec	feet/sec^2
`English (kts)`	Pound-force	Foot-pound	Slugs	Slug ft^2	nautical mile	knots	knots/sec

Modeling a Satellite Constellation

Up to this point we have modeled a single spacecraft with the Spacecraft Dynamics block. However, the block can also be configured to model a constellation of satellites/spacecraft. The number of spacecraft being modeled is determined by the size of the initial conditions provided. If more than one value is provided for a parameter in the **Mass**, **Orbit**, or **Attitude** tabs, the block outputs a constellation of satellites. Any parameter that has a single value provided is expanded and applied to all satellites in the constellation. For example, if a single value is provided for all parameters on the block except **True anomaly** which contains 6 values, a constellation of 6 satellites is created, varying true anomaly only.



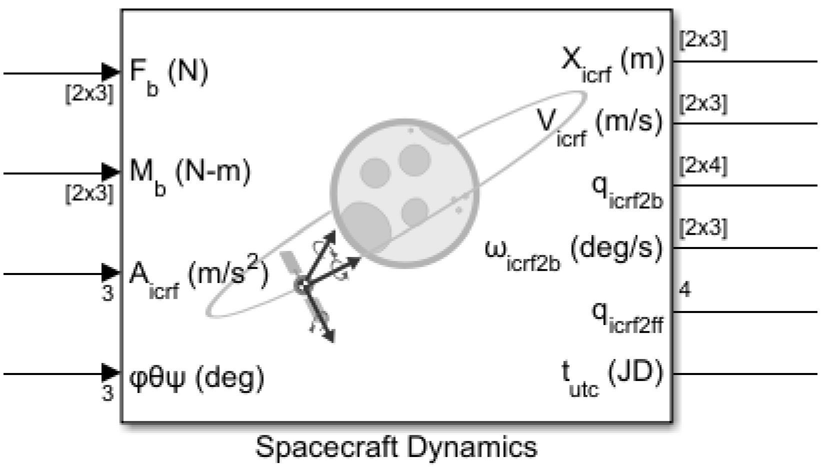
This behavior applies to all spacecraft initial conditions (all dialog boxes in the **Mass**, **Orbit**, or **Attitude** tabs). Above, initial condition parameter in the **Mass**, **Orbit**, or **Attitude** tabs must contain a single value expanded to all satellite or 6 individual values, one for each satellite.



The same expansion behavior also applies to block input ports. All input ports support expansion *expect* **Moon libration angles** $\varphi\theta\psi$ (when **Central body** is `Moon`) and spin-axis **Right ascension, declination, and rotation angle** $\alpha\delta W$ (when **Central body** is `Custom`). Moon libration angles and spin-axis orientation inputs are time-dependent values, and therefore always apply to all spacecraft being modeled. All other ports accept a single value expanded to all spacecraft being modeled, or individual values applied to each spacecraft (6, in the above example).

Modeling a Lunar Orbit

To demonstrate this port expansion behavior, consider a new scenario in which we have twin lunar orbiters separated along their orbit track by 200km. Each satellite operates independently of the other, so different forces and moments are applied to each. However, we want to include the gravitational impact of Earth as a perturbing acceleration on both satellites. We assume that the difference in gravitational acceleration due to Earth in a lunar orbit across 200km is negligible. Our resulting block is shown below.



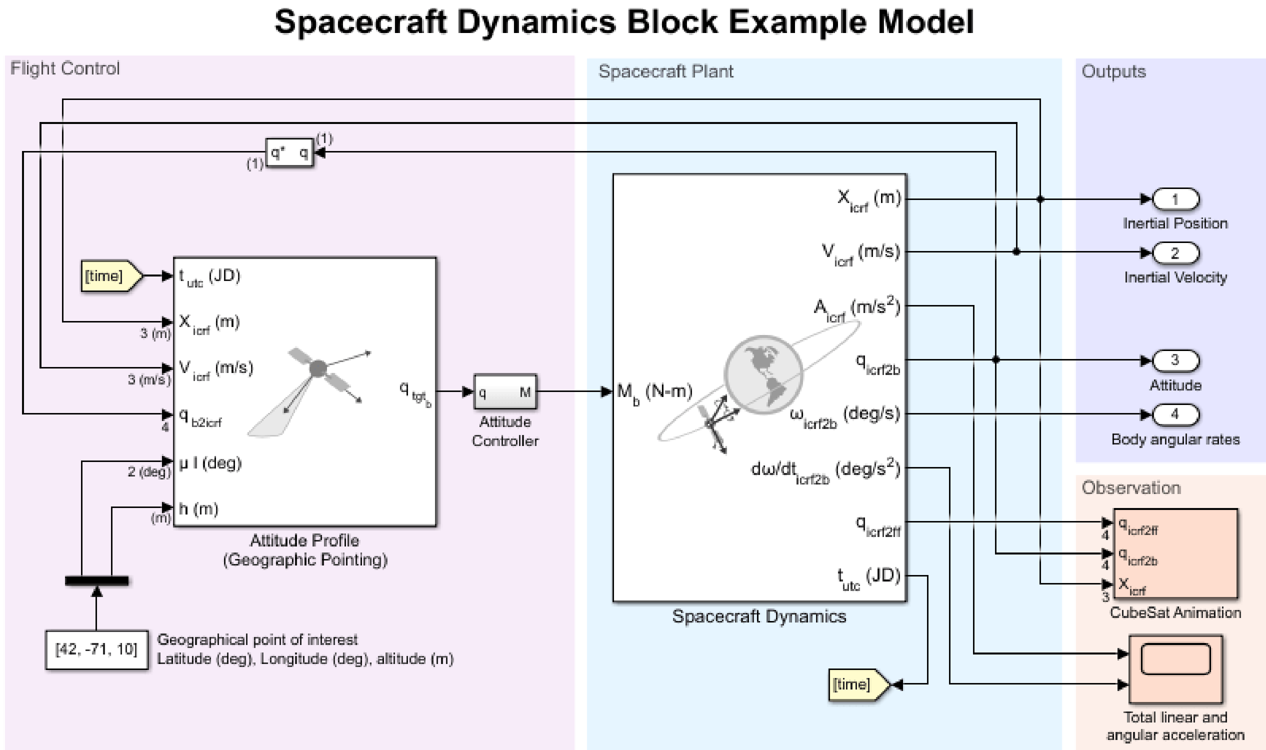
There are separate force and moment input values for each satellite, however a single external acceleration input is expanded and applied to both spacecraft. As stated above, Moon libration angles $\varphi\theta\psi$ are always spacecraft-independent.

The state outputs from the blocks always match the total number of spacecraft being modeled, where rows correspond with individual spacecraft. There are also two time-dependent outputs from the block, the current time t_{utc} and the transformation from inertial frame to fixed frame q_{icrf2ff}.

Simulink Model Example

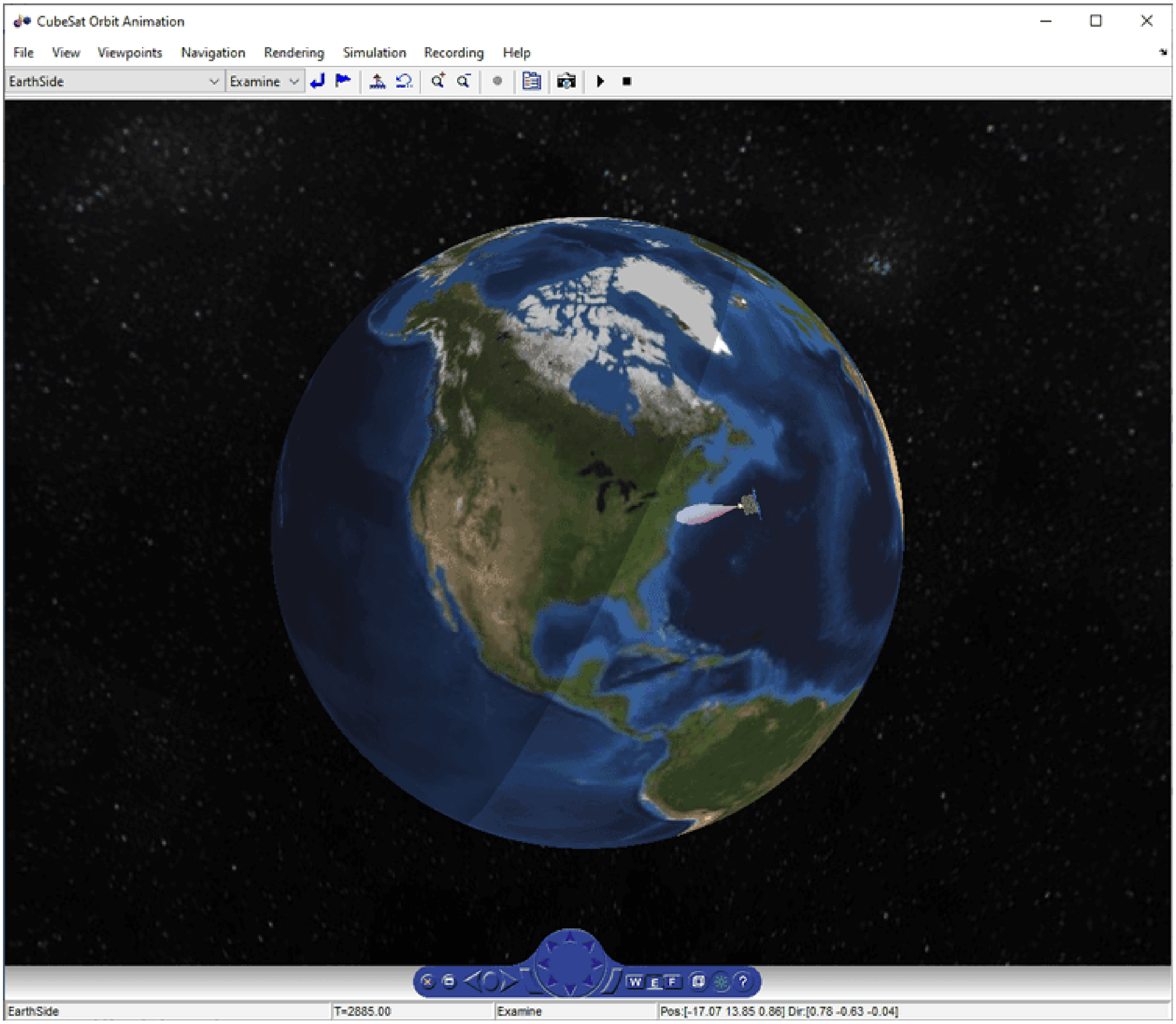
Now, explore an example model that uses the Spacecraft Dynamics block to simulate an Earth observation satellite.

```
mdl = "SpacecraftDynamicsBlockExampleModel";
open_system(mdl);
```

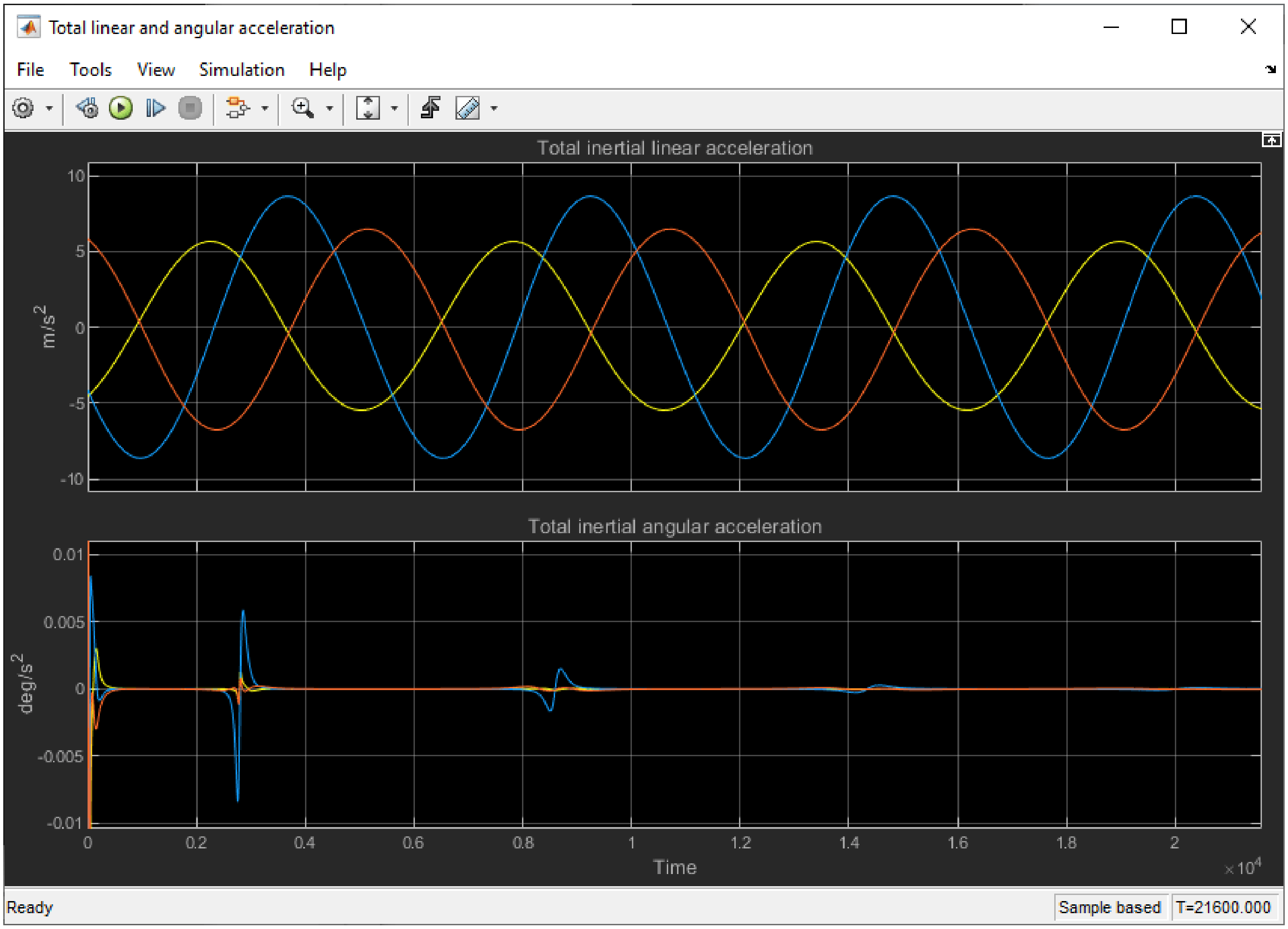


The satellite is in a near-circular, low Earth orbit (LEO) at an altitude of approximately 500km. For orbit propagation, we use Earth spherical harmonic model EGM2008 with degree set to 120. We use Earth orientation parameter data from default file `aeroiersdata.mat`, which is included in the Aerospace Toolbox. The satellite mass properties are fixed, with mass equal to 1kg and a simple inertia tensor of $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The mission start date is January 1, 2020, 12:00:00, and runs for 6 hours.

To provide attitude control, we use the **`Attitude Profile`** block from the Aerospace Blockset, connected to a simple PD controller. Our desired attitude aligns the satellite body-*z* axis with geographic coordinates $[42^\circ, -71^\circ]$ at an altitude of 10m. For our secondary constraint, we align the body-*x* axis with the *y*-axis of the local-vertical, local-horizontal (LVLH) frame. In a (near) circular orbit, the *y*-axis of the LVLH frame points in the direction of the travel of the satellite. This alignment keeps our satellite pointing "forward" as we sweep over our geographic point of interest on each pass. You can also use the **`Attitude Profile`** block to align the satellite with Earth nadir, a different geographical location, a celestial body from JPL Ephemerides DE405, or any custom vector provided as an input to the block.



Included in the model is a Simulink 3D Animation™ world configured to visualize a 1U CubeSat. This block is commented out by default because it requires Simulink 3D Animation. To enable the visualization and change visualization properties, double-click the block.



The model also has linear and angular acceleration outputs from the Spacecraft Dynamics block connected to a scope. Do not use these outputs as part of a simulation loop. Our inertial linear acceleration is smooth throughout the simulation, which is expected as we are not performing any translational (delta-V) maneuvers. In the angular acceleration plot, we observe that passes over our geographical point of interest coincide with larger acceleration values. When we pass directly over the point of interest (the first pass), the change in angular velocity required is much larger than when we pass over the point of interest at a shallower angle (subsequent passes).

Block Equations

We now explore the equations implemented by the block to better understand how the block calculates output values at each timestep.

Translational System Equations

Translation motion is governed by:

$$\vec{a}_{\text{icrf}} = \vec{a}_{\text{central body gravity}} + \text{body2inertial}\left(\frac{\vec{F_b}}{m}\right) + \vec{a}_{\text{applied}}$$

$$\vec{a}_{\text{icrf}} \xRightarrow[\text{integrate}]{} \vec{v}_{\text{icrf}}, \vec{r}_{\text{icrf}}$$

where:

$\vec{a}_{\text{central body gravity}}$ is the central body gravity based on the current block parameter selections.

$\vec{a}_{\text{applied}}$ is the user-defined acceleration provided to the block external acceleration input port.

$\overrightarrow{F_b}$ is the body force in the Body coordinate system, with respect to the ICRF frame (inertial).

m is the current spacecraft mass (see mass equations on page 9-137 below).

body2inertial( ) is the transformation from the rotating body-fixed coordinate system to the inertial ICRF coordinate system, resulting in the following acceleration contribution from forces:
$\text{body2inertial}\left(\frac{\overrightarrow{F_b}}{m}\right) = \vec{a}_{\text{icrf}_{\text{forces}}} = \text{quatrotate}\left(q_{\text{b2icrf}}, \vec{a}_b\right)$

where:

$$\vec{a}_b = \frac{\vec{F}_b}{m} = \frac{\vec{F}_{b_{\text{input}}} + \dot{m}\vec{v}_{\text{re}}}{m}$$

$$\vec{r}_b = \text{quatrotate}\left(q_{\text{icrf2b}}, \vec{r}_{\text{icrf}}\right)$$

$\vec{F}_{b_{\text{input}}}$ is the force provided to the block body forces input port.

$\vec{v}_{\text{re}}$ is the relative velocity at which the mass flow ($\dot{m}$) is ejected from or added to the body in the Body coordinate system, with respect to the Body frame.

q_{icrf2b} is the passive quaternion rotation of the body with respect to the inertial ICRF frame.

$\vec{\omega}_{\text{icrf2b}}$ is the angular velocity of the body with respect to the inertial ICRF frame.

Rotational System Equations

Rotational motion is governed by:

$$\dot{\vec{\omega}}_{\text{icrf2b}} = \left[\overrightarrow{\mathrm{M}_b} - \vec{\omega}_{\text{icrf2b}} \times \left(I_{mom}\vec{\omega}_{\text{icrf2b}}\right) - \dot{I}_{mom}\vec{\omega}_{\text{icrf2b}}\right]\text{inv}(I_{mom})$$

$$\dot{\vec{\omega}}_{\text{icrf2b}} \xRightarrow[\text{integrate}]{} \vec{\omega}_{\text{icrf2b}}, \overrightarrow{q_{\text{icrf2b}}}$$

where:

$I_{mom} = \begin{bmatrix} I_{\text{xx}} & -I_{\text{xy}} & -I_{\text{xz}} \\ -I_{\text{yx}} & I_{\text{yy}} & -I_{\text{yz}} \\ -I_{\text{zx}} & -I_{\text{zy}} & I_{\text{zz}} \end{bmatrix}$ is the inertia tensor with respect to the body origin (see Mass section on page 9-137 below).

$\dot{I}_{mom}$ is the rate of change of the inertia tensor (see Mass section on page 9-137 below).

inv( ) is the 3x3 matrix inverse.

$\overrightarrow{\mathrm{M}_b} = \overrightarrow{M}_{b_{\text{input}}} + M_{b_{\text{gravity gradient}}}$ is the total body moment, comprised of the value provided to the block body moments input port and the internally calculated gravity gradient torque:

$$M_{b_{\text{gravity gradient}}} = \frac{3\mu}{r_b^5}\overrightarrow{r}_b \times I_{mom}\overrightarrow{r}_b$$

μ is the standard gravitation parameter of the central body.

The integration of the rate of change of the quaternion vector is calculated as:

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} 0 & \omega_b(1) & \omega_b(2) & \omega_b(3) \\ -\omega_b(1) & 0 & -\omega_b(3) & \omega_b(2) \\ -\omega_b(2) & \omega_b(3) & 0 & -\omega_b(1) \\ -\omega_b(3) & -\omega_b(2) & \omega_b(1) & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

The Aerospace Toolbox and Aerospace Blockset use quaternions that are defined using the scalar-first convention.

Mass

The mass m, mass flow rate $\dot{m}$, inertia tensor I_{mom}, and rate of change of the inertia tensor $\dot{I}_{\text{mom}}$used in the above system equations are determined depending on the current parameter selections in the **Mass** tab.

`Fixed`

This option models the spacecraft as a fixed mass rigid body.

m is the mass provided for parameter **Mass** on the **Mass** tab.

$\dot{m}$ equals zero.

I_{mom} is the inertia tensor provided for parameter **Inertia tensor** on the **Mass** tab.

$\dot{I}_{\text{mom}}$ equals zero.

`Simple variable`

This option models the spacecraft as a simple, variable-mass rigid body.

m is the mass bounded between m_{full} and m_{empty}, calculated by integrating $\dot{m}$.

$\dot{m}$ is provided to the block's **`dm/dt`** input port.

$$I_{\text{mom}} = \frac{I_{\text{full}} - I_{\text{empty}}}{m_{\text{full}} - m_{\text{empty}}}\left(m - m_{\text{empty}}\right) + I_{\text{empty}}$$

$$\dot{I}_{\text{mom}} = \frac{I_{\text{full}} - I_{\text{empty}}}{m_{\text{full}} - m_{\text{empty}}}\dot{m}$$

`Custom variable`

This option models the spacecraft as a variable-mass rigid body, providing the highest level of configurability.

m is provided to the block **`m`** input port.

$\dot{m}$ is provided to the block **`dm/dt`** input port when **Include mass flow relative velocity** is enabled, otherwise the value is not needed by the system equations.

I_{mom} is provided to the block **`I`** input port.

$\dot{I}_{\text{mom}}$ is provided to the block **`dI/dt`** input port.

Central Body Gravity

The acceleration due to central body gravity $\vec{a}_{\text{central body gravity}}$ is calculated depending on the current parameter selections in the **Central Body** tab. For gravity models that include nonspherical acceleration terms (`Oblate ellipsoid (J2)` and `Spherical harmonics`), nonspherical gravity is computed in the fixed-frame coordinated system (ITRF, in the case of Earth). Numerical integration, however, is always performed in the inertial ICRF coordinate system. Therefore, at each timestep, position and velocity states are transformed into the fixed-frame, nonspherical gravity is calculated in the fixed-frame, and the resulting acceleration is then transformed into the inertial frame. In the inertial frame, the resulting acceleration is summed with the other acceleration terms and double-integrated to find velocity and position.

`Point-mass` (available for all central bodies)

This option treats the central body as a point mass, including only the effects of spherical gravity using Newton's law of universal gravitation.

$$\vec{a}_{\text{central body gravity}} = -\frac{\mu}{r_{\text{icrf}}^2}\frac{\vec{r}_{\text{icrf}}}{r_{\text{icrf}}}$$

`Oblate ellipsoid (J2)` (available for all central bodies)

In addition to spherical gravity, this option includes the perturbing effects of the second-degree, zonal harmonic gravity coefficient, J_2, accounting for the oblateness of the central body. J_2 accounts for most of central body gravitational departure from a perfect sphere.

$$\vec{a}_{\text{central body gravity}} = -\frac{\mu}{r_{\text{icrf}}^2}\frac{\vec{r}_{\text{icrf}}}{r_{\text{icrf}}} + \text{fixed2inertial}\left(\vec{a}_{\text{nonspherical}}\right)$$

where:

$$\overrightarrow{a}_{\text{nonspherical}} = \left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{\text{ff}k}}{r^2\sqrt{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{\text{ff}i}\right\}i + \left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{\text{ff}k}}{r^2\sqrt{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{\text{ff}j}\right\}j + \left\{\frac{1}{r}\left(\frac{\partial}{\partial r}U\right)r_k + \frac{\sqrt{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}}{r^2}\frac{\partial}{\partial \phi}U\right\}k$$

given the partial derivatives in spherical coordinates:

$$\frac{\partial}{\partial r}U = \frac{3\mu}{r^2}\left(\frac{R_{\text{cb}}}{r}\right)^2 P_{2,0}[\sin(\phi)]J_2$$

$$\frac{\partial}{\partial \phi}U = -\frac{\mu}{r}\left(\frac{R_{\text{cb}}}{r}\right)^2 P_{2,1}[\sin(\phi)]J_2$$

where:

ϕ and λ are the satellite geocentric latitude and longitude

$P_{2,0}$ and $P_{2,1}$ are associated Legendre functions

R_{cb} is the central body equatorial radius

fixed2inertial( ) converts fixed-frame position, velocity, and acceleration into the ICRF coordinate system with origin at the center of the central body, accounting for centrifugal and Coriolis acceleration. For more information about the fixed and inertial coordinate systems used for each central body, see the Coordinate Frames section of the Spacecraft Dynamics block reference page. The fixed-frame coordinate frame used for Earth is the ITRF.

`Spherical Harmonics` (available for Earth, Moon, Mars, Custom)

This option adds increased fidelity by including higher-order perturbation effects accounting for zonal, sectoral, and tesseral harmonics. For reference, the second-degree zeroth order zonal harmonic J_2 is $-C_{2,0}$. The Spherical Harmonics model accounts for harmonics up to max degree $l = l_{\text{max}}$, which varies by central body and geopotential model.

$$\overrightarrow{a}_{\text{central body gravity}} = -\frac{\mu}{r^2}\frac{\overrightarrow{r_{\text{icrf}}}}{r} + \text{fixed2inertial}\left(\overrightarrow{a}_{\text{nonspherical}}\right)$$

where:

$$\overrightarrow{a}_{\text{nonspherical}} = \left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{\text{ff}k}}{r^2\sqrt{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{\text{ff}i} - \left[\frac{1}{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}\frac{\partial}{\partial \lambda}U\right]r_{\text{ff}j}\right\}i + \left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{\text{ff}k}}{r^2\sqrt{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}}\frac{\partial}{\partial \phi}U\right]r_{\text{ff}j} + \left[\frac{1}{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}\frac{\partial}{\partial \lambda}U\right]r_{\text{ff}i}\right\}j + \left\{\frac{1}{r}\left(\frac{\partial}{\partial r}U\right)r_{\text{ff}k} + \frac{\sqrt{r_{\text{ff}i}{}^2 + r_{\text{ff}j}{}^2}}{r^2}\frac{\partial}{\partial \phi}U\right\}k$$

given the following partial derivatives in spherical coordinates:

$$\frac{\partial}{\partial r}U = -\frac{\mu}{r^2}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\text{cb}}}{r}\right)^l (l+1)P_{l,m}[\sin(\phi)]\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\}$$

$$\frac{\partial}{\partial \phi}U = \frac{\mu}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\text{cb}}}{r}\right)^l \{P_{l,m+1}[\sin(\phi)] - (m)\tan(\phi)P_{l,m}[\sin(\phi)]\}\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\}$$

$$\frac{\partial}{\partial \lambda}U = \frac{\mu}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\text{cb}}}{r}\right)^l (m)P_{l,m}[\sin(\phi)]\{S_{l,m}\cos(m\lambda) - C_{l,m}\sin(m\lambda)\}$$

$P_{l,m}$ are associated Legendre functions.

$C_{l,m}$ and $S_{l,m}$ are the un-normalized harmonic coefficients.

References

[1] Vallado, David. Fundamentals of Astrodynamics and Applications, 4th ed. Hawthorne, CA: Microcosm Press, 2013.

[2] Vepa, R., Dynamics and Control of Autonomous Space Vehicles and Robotics, University Printing House, New York, NY,USA, 2019.

[3] Stevens, B. L., F. L. Lewis, Aircraft Control and Simulation, Second Edition, John Wiley & Sons, Hoboken NJ, 2003.

[4] Gottlieb, R. G., "Fast Gravity, Gravity Partials, Normalized Gravity, Gravity Gradient Torque and Magnetic Field: Derivation, Code and Data," Technical Report NASA Contractor Report 188243, NASA Lyndon B. Johnson Space Center, Houston, Texas, February 1993.

[5] Konopliv, A. S., S. W. Asmar, E. Carranza, W. L. Sjogen, D. N. Yuan., "Recent Gravity Models as a Result of the LunarProspector Mission, Icarus", Vol. 150, no. 1, pp 1–18, 2001.

[6] Lemoine, F. G., D. E. Smith, D.D. Rowlands, M.T. Zuber, G. A. Neumann, and D. S. Chinn, "An improved solution of the gravity field of Mars (GMM-2B) from Mars Global Surveyor", Journal Of Geophysical Research, Vol. 106, No. E10, pp 23359-23376, October 25, 2001.

[7] Seidelmann, P.K., Archinal, B.A., A'hearn, M.F. et al. Report of the IAU/IAG Working Group on cartographic coordinates and rotational elements: 2006. Celestial Mech Dyn Astr 98, 155–180 (2007).

[8] Standish, E. M., "JPL Planetary and Lunar Ephemerides, DE405/LE405", JPL IOM 312.F-98-048, Pasadena, CA,1998.

See Also

Spacecraft Dynamics

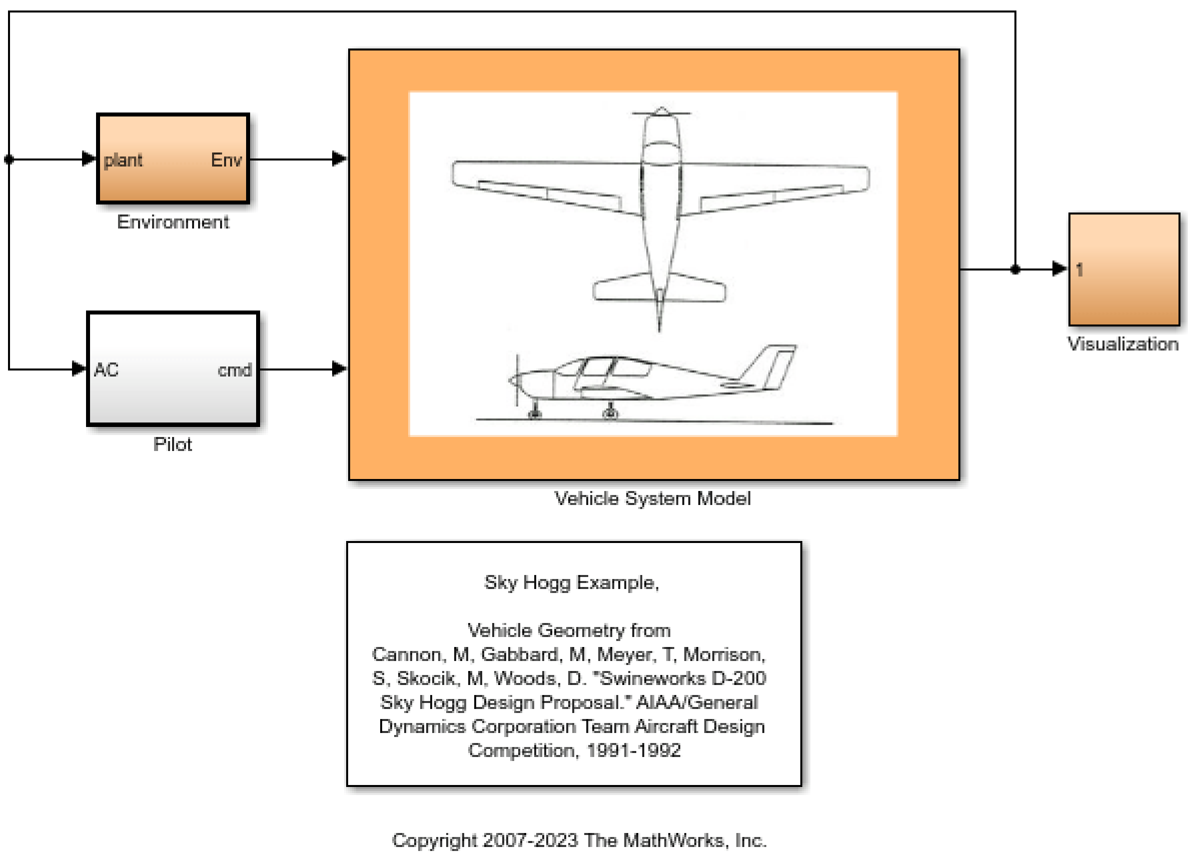
Using Unreal Engine Visualization for Airplane Flight

This example shows how to add Unreal Engine® visualization using the Aerospace Simulation 3D library blocks. In this example, the Sky Hogg airplane flies over the prebuilt airport scene. To see the final model incorporating 3D visualization, open `SkyHoggSim3DExampleModel`.

Note: This example is not supported in Simulink Online.

Start with Sky Hogg Example Model

Open the model used for the Lightweight Airplane Design example, `asbSkyHogg`.



This model is set up for visualization using FlightGear in the **Visualization** subsystem. This example shows how to replace that implementation using Unreal Engine®.

Open the **Visualization** subsystem and delete everything except for In1 and Bus Selector1.

Adding Simulation 3D Animation Blocks

Use the library browser to go to Aerospace Blockset > Animation > Simulation 3D.

To the model:

1 Add the blocks Simulation 3D Scene Configuration, Simulation 3D Aircraft, and Simulation 3D Sky Hogg Pack.
2 Add a Scope block and attach it to the aircraft Altitude port.
3 Terminate the Simulation 3D Aircraft WoW port.

The WoW (Weight on Wheels) port returns a logical true if either the left or right main gear tire is on a surface (i.e. its altitude is zero) or false otherwise. Since this example has a flying aircraft, this port is not used.

Setting Up Simulation 3D Aircraft Block

Double-click to open the Simulation 3D Aircraft block to set up values on the **Aircraft Parameters** and **Altitude Sensor** tabs.

Aircraft Parameters tab

In the **Aircraft Parameters** tab:

- Set the **Type** parameter to Sky Hogg.
- Select the desired color.
- Leave the default name of `SimulinkVehicle1`.
- Leave the **Sample time** value of `-1` to allow the block to use the sample time in the Simulation 3D Scene Configuration block.

The initial conditions for the input ports are given in the **Initial Values** group. Each must be an 11-by-3 array. The aircraft component associated with each row for the Sky Hogg is as follows.

1	2	3	4	5	6
BODY	PROPELLER	RUDDER	ELEVATOR	LEFT_AILERON	RIGHT_AILERON

7	8	9	10	11
FLAPS	NOSE_WHEEL_STRUT	NOSE_WHEEL	LEFT_WHEEL	RIGHT_WHEEL

For this example, change the **Initial translation** value to `[0 0 -2000; zeros(10, 3)]` and leave the **Initial rotation** value at `zeros(11, 3)`. The -2000 meter *Z* value is the negative of the initial altitude (NED).

Altitude Sensor tab

The altitude sensor is optional and can be turned on and off by the **Enable altitude and WoW sensors** check box on this tab. The sensor works by sending ray traces vertically down from the aircraft body and each of its wheels. Altitude is only sensed if an object is hit by the rays, which are of the prescribed finite length. The *Z* offset values place the starting point of each ray at the given vertical (downward) distance from the aircraft body origin or wheel centers. For example, if the *Z* offset value entered for the **Front tire radius (m)** is the actual front tire radius for the aircraft mesh selected, then the returned second altitude value is zero when the aircraft front gear tire sits on the pavement.

Leave the default settings in place for now.

Setting Up Simulation 3D Scene Configuration Block

Check the configuration of the Simulation 3D Scene Configuration block. It should have **Scene source** set to `Default Scenes`, with the `Airport` scene selected. For the **Scene view**, use the name entered in the Simulation 3D Aircraft block, which by default is `SimulinkVehicle1`. A **Sample time** of `1/60` or similar is fine; use a smaller value for a higher frame rate. To experiment with the weather, see the controls on the **Weather** tab. Note that weather in Unreal Engine® is currently just a visual sky effect; there are no actual wind vectors or forces, for example.

Connecting Simulation 3D Aircraft to Sky Hogg Translation and Rotation

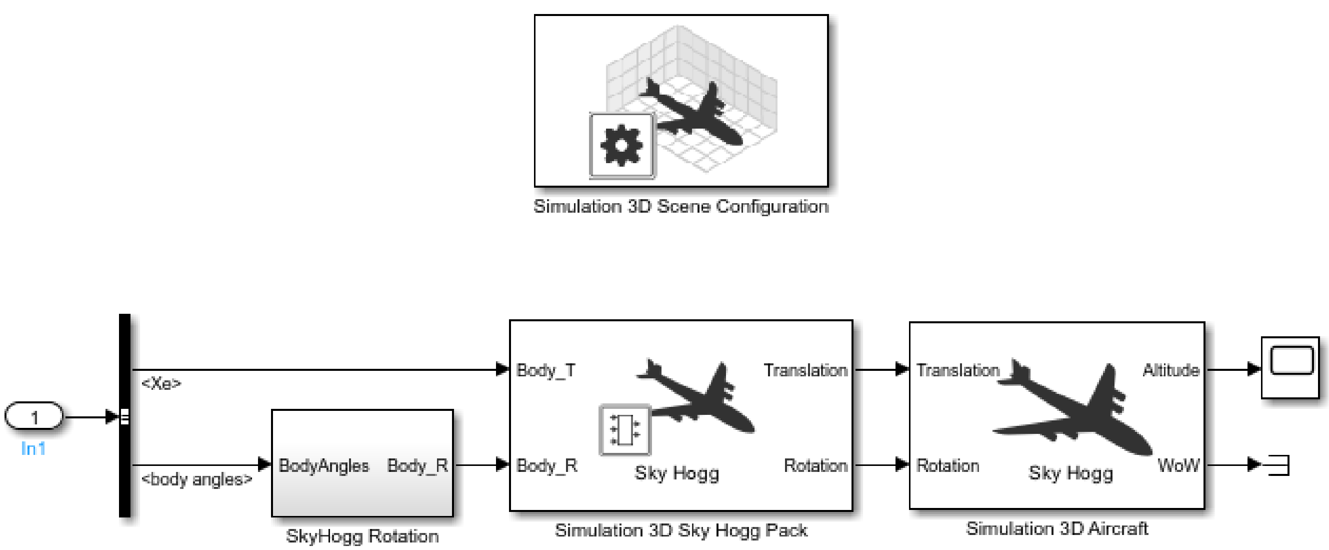
The remaining step is to configure the Translation and Rotation port inputs to the Simulation 3D Aircraft block. These ports expect 11-by-3 array input at every time step when using Sky Hogg. See the Simulation 3D Aircraft block reference page for a full description. Since control surface motions are not provided by the model, change just the values of BODY.

Reconfigure the bus selector to output just `Xe` and `body angles`.

Connect the Simulation 3D Sky Hogg Pack block output ports to the input ports of the Simulation 3D Aircraft block. Next connect the `Xe` signal from the bus selector to the `Body_T` input port of the pack block.

For rotation, the `body angles` bus must be reassembled into a 1-by-3 vector. Create a subsystem to do this. Use a Bus Selector to obtain the three angles from the input and feed them into a Vector Concatenate block, then use a Reshape block to output a 2D row vector.

The **Visualization** subsystem should look like this:



Simulation

Model is ready to run.

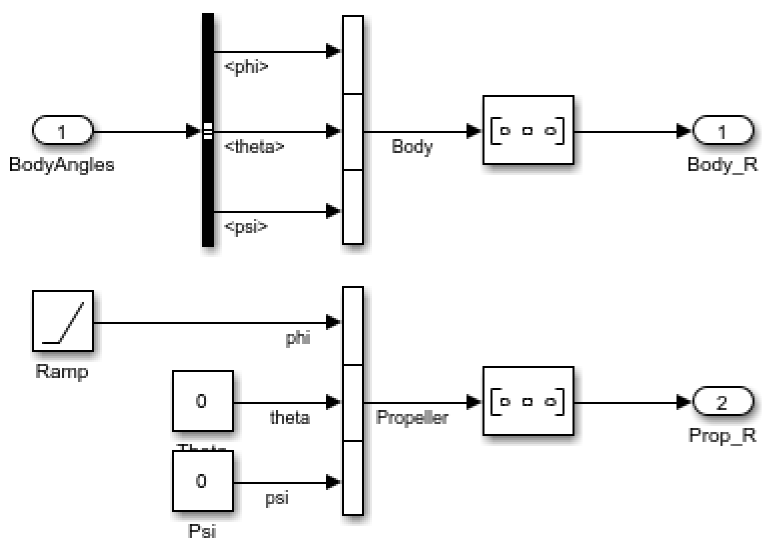
- After pressing the **Run** button, allow a few seconds for the 3D visualization window to initialize.
- You should now see the airplane flying over the airport.
- Once it is simulating, you can switch between camera views by first left-clicking inside the 3D window, then using the numbers keys *0* through *9* to choose between ten preconfigured camera positions. For more information on camera views, see the **Run Simulation** section in **Customize Scenes Using Simulink and Unreal Editor**.

Improving the Visualization to Simulation Interaction

Since the height change is so small (50 meters), it is difficult to see the altitude increase in the 3D window. For illustrative purposes, add a Gain block to increase the translation *Z* values.

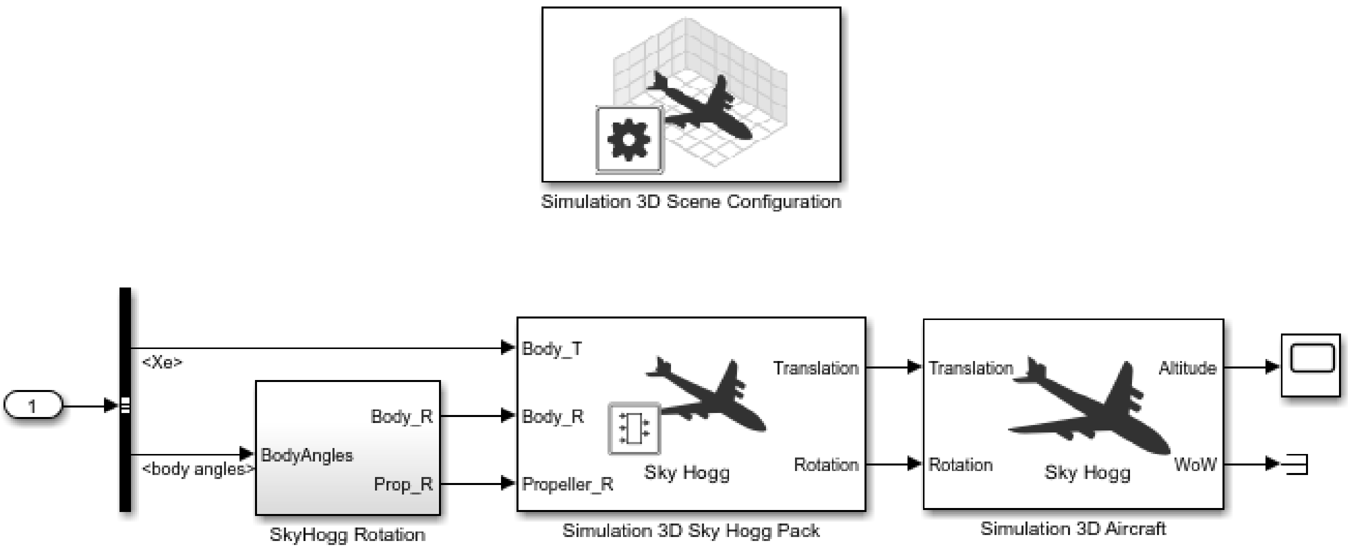
Adding Propeller Rotation

This scene is not very realistic since the propeller is not turning. Propeller rotation is not typically calculated in the model, but you can choose a rotation rate for it. To rotate it at 1500 RPM, or 157 radians per second, add a Ramp block for the roll (*phi*) angle. The modified **Sky Hogg Rotation** subsystem should look like this.



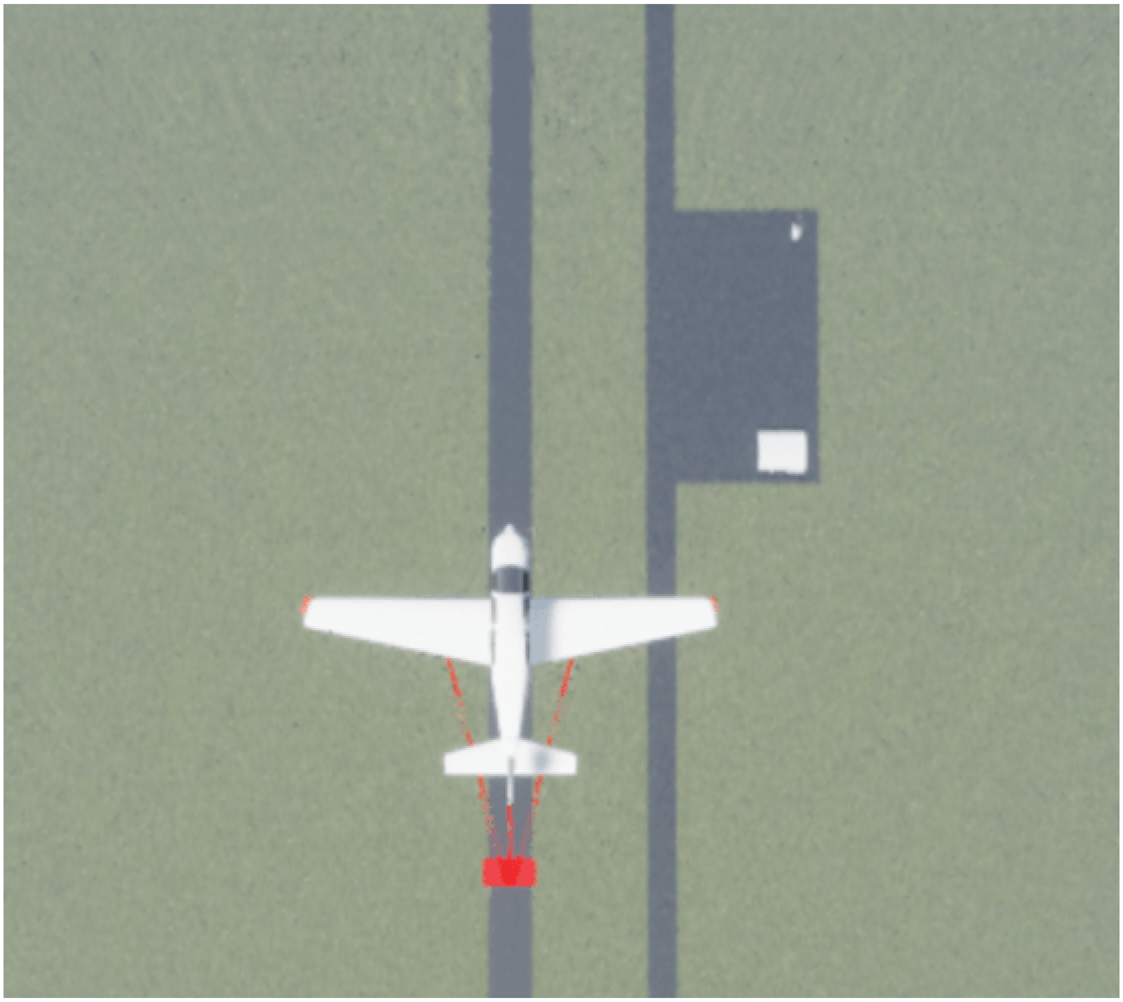
Open the pack block mask and select the **Propeller rotation** check box, then click **OK** to close the mask. The **Propeller_R** port appears. Connect the **Prop_R** port from **Sky Hogg Rotation** to this port.

The final **Visualization** subsystem should look like this:



Altitude Sensor Rays

At first glance, the altitude sensor Scope block does not appear to be working (returning `-1` values). This is because the altitude is greater than the length of the rays. Open the Simulation 3D Aircraft block mask and change the **Length of rays (m)** to `2500`. If you want to see the rays, select the **Show sensor rays in viewer** check box. Run the simulation again. The altitudes output in the two scopes validates that it is indeed at the prescribed location. If visible sensor rays are enabled, then they are colored red since they are hitting the ground. Without changing the ray length, the rays are colored green (if made visible) because they do not reach the ground.



Updated Simulation 3D Visualization Model

All of these steps have been completed for you in the following example model.

```
mdl = "SkyHoggSim3DExampleModel";
open_system(mdl);
```

See Also

Simulation 3D Aircraft | Simulation 3D Scene Configuration

Related Examples

- Visualization Techniques with Aerospace Blockset Video

Developing the Apollo Lunar Module Digital Autopilot

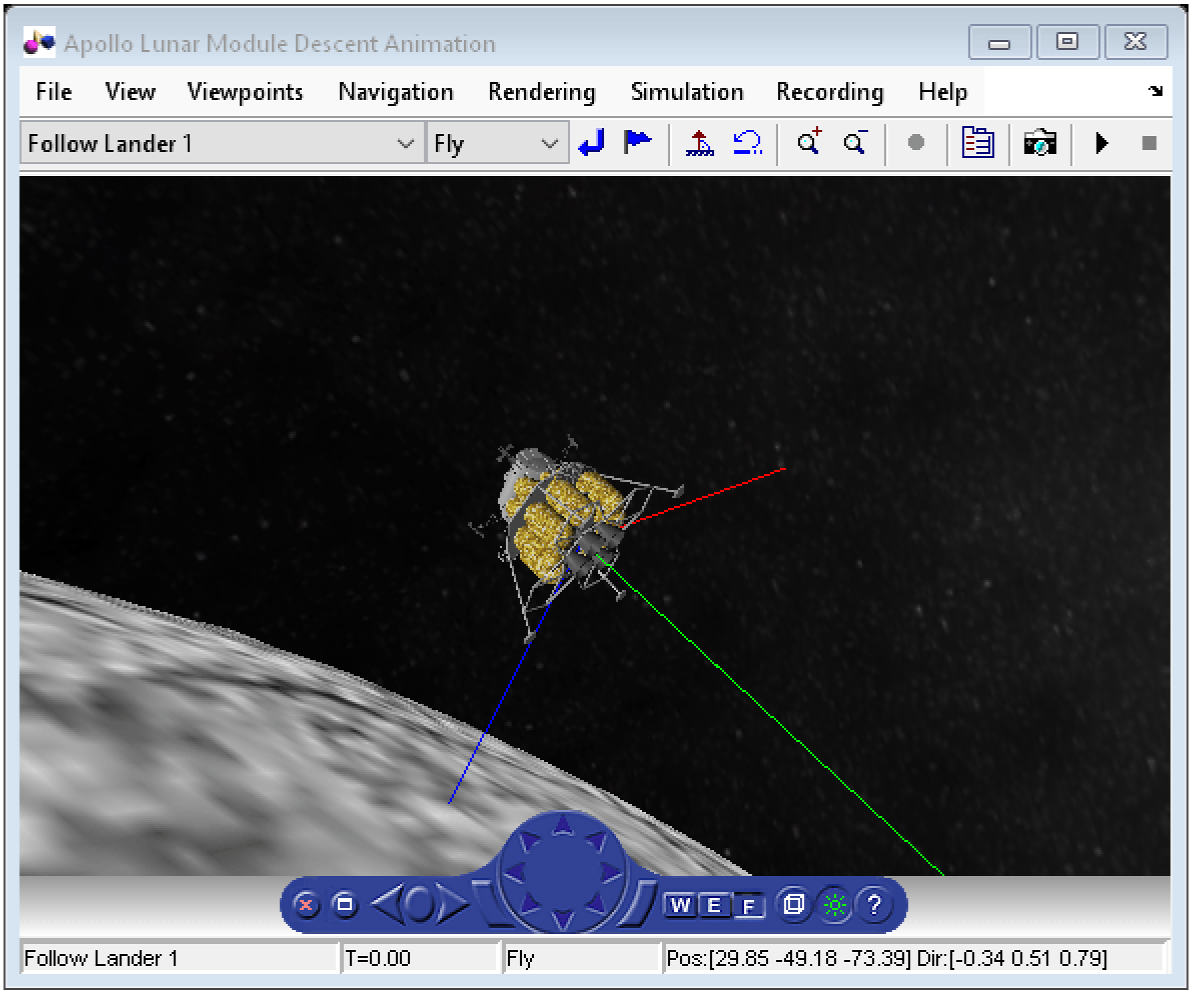
"Working on the design of the Lunar Module digital autopilot was the highlight of my career as an engineer. When Neil Armstrong stepped off the LM (Lunar Module) onto the moon's surface, every engineer who contributed to the Apollo program felt a sense of pride and accomplishment. We had succeeded in our goal. We had developed technology that never existed before, and through hard work and meticulous attention to detail, we had created a system that worked flawlessly." -Richard J. Gran, *The Apollo 11 Moon Landing: Spacecraft Design Then and Now.*

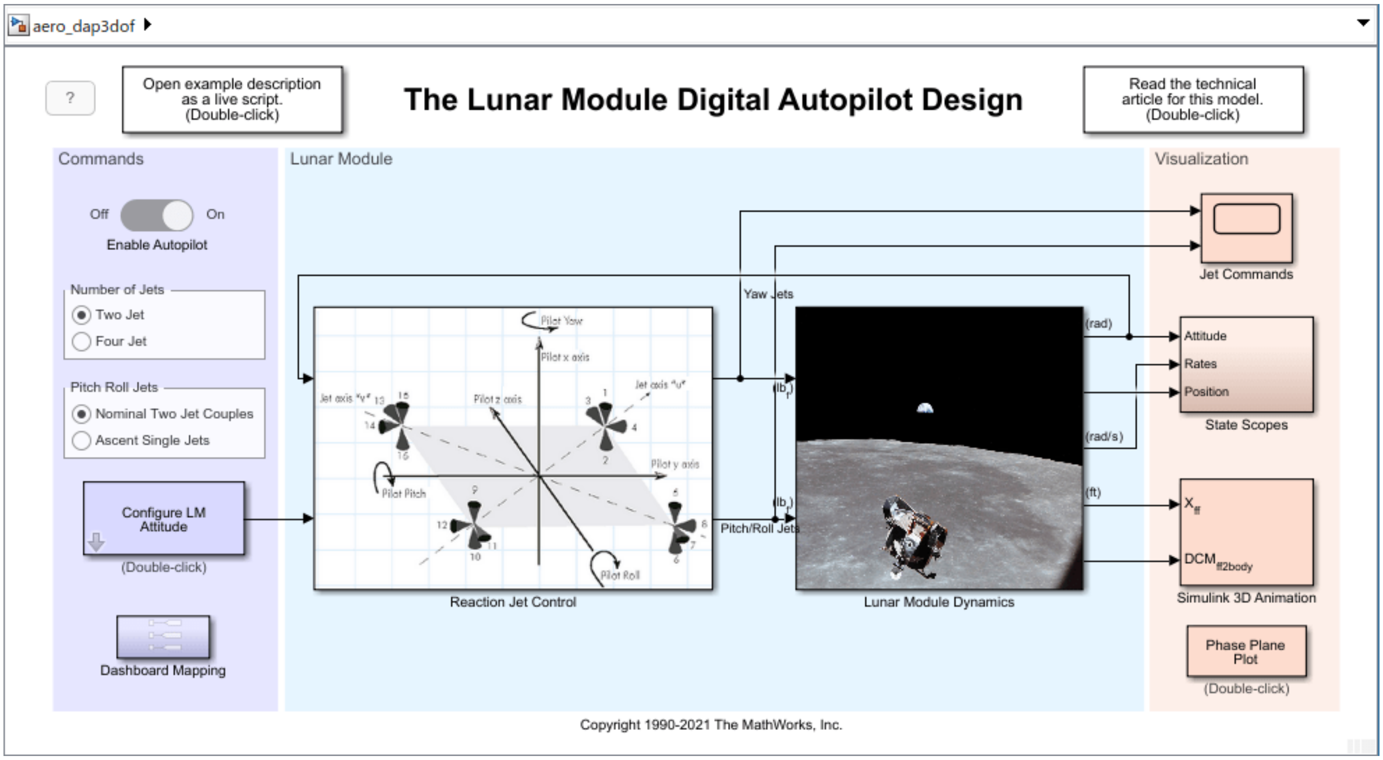
This example shows how Richard and the other engineers who worked on the Apollo Lunar Module digital autopilot design team could have done it using Simulink® and Aerospace Blockset™ if they had been available in 1961.

Model Description

Developing the autopilot in Simulink takes a fraction of the time it took for the original design of the Apollo Lunar Module autopilot.

```
if ~bdIsLoaded("aero_dap3dof")
    open_system("aero_dap3dof");
end
```


Apollo Lunar Module Descent Animation
File View Viewpoints Navigation Rendering Simulation Recording Help
Follow Lander 1
Fly
W E F
Follow Lander 1
T=0.00
Fly
Pos:[29.85 -49.18 -73.39] Dir:[-0.34 0.51 0.79]

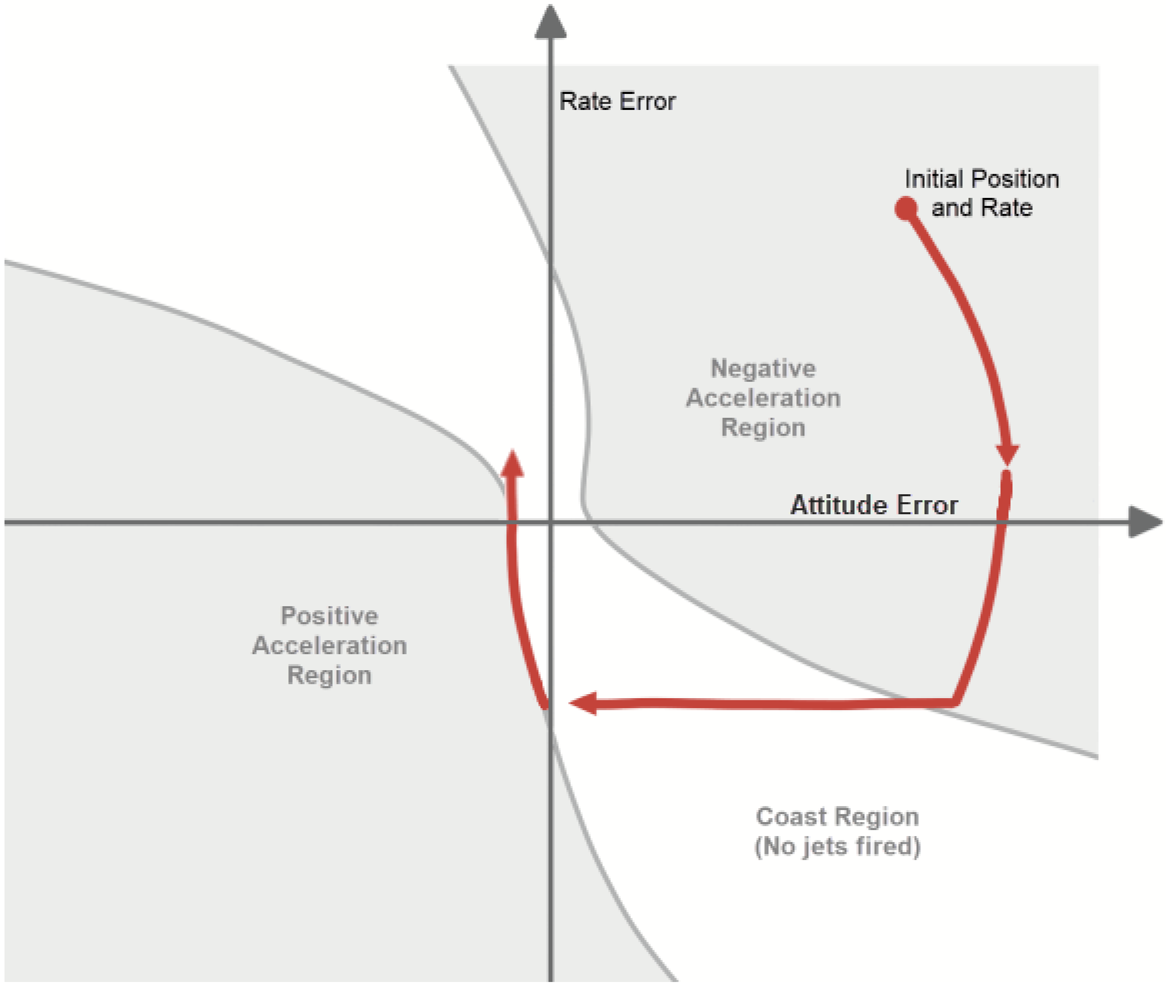


The Reaction Jet Control subsystem models the digital autopilot design proposed (and implemented) by MIT Instrumentation Laboratories (MIT IL), now called Draper Laboratory. A Stateflow® chart in the model specifies the logic that implements the phase-plane control algorithm described in the technical article *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*. Depending on which state of the chart is executing, the Stateflow diagram is in either a `Fire_region` or a `Coast_region`. The Stateflow chart determines whether to transition to another state and then computes which reaction jets to fire.

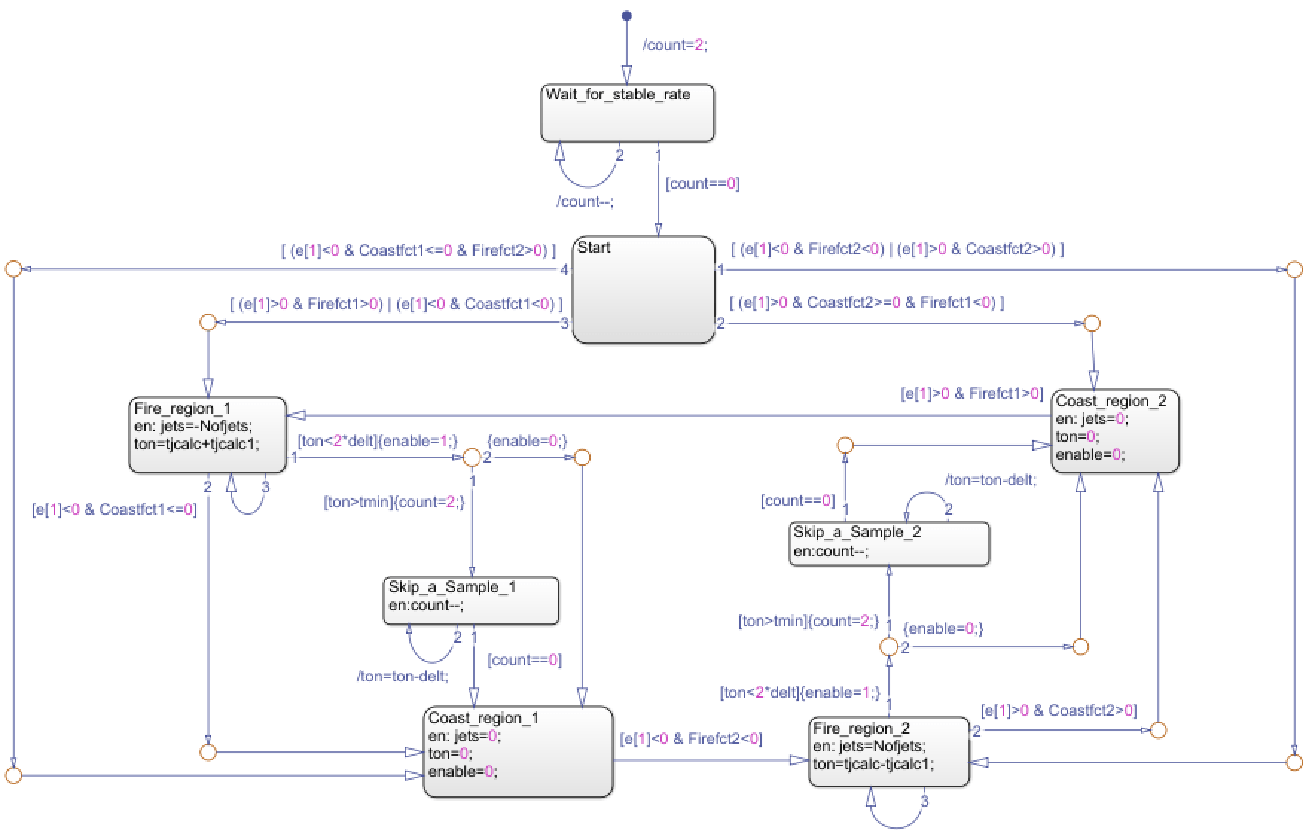
To control the firing of the reaction jets, engineers developed an optimal controller that minimizes the weighted mix of time and fuel based on errors in the attitude and rate. The control can be visualized as three distinct regions in the attitude-rate space:

- Region 1: Fire the jets to produce negative acceleration
- Coast region: Do not fire the jets
- Region 2: Fire the jets to produce positive acceleration

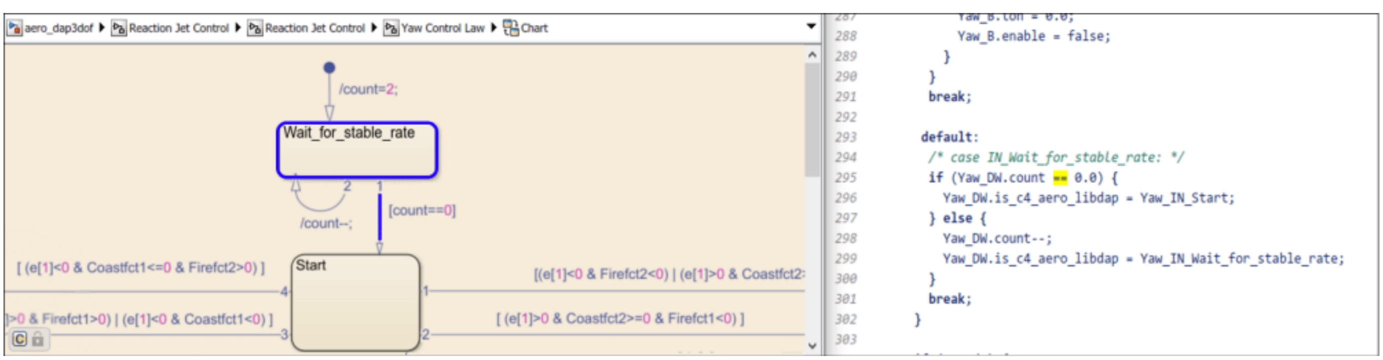
The lunar module begins its journey at the Initial Position and Rate and fires jets or coasts to arrive at, and maintains, its location in the Coast Region.



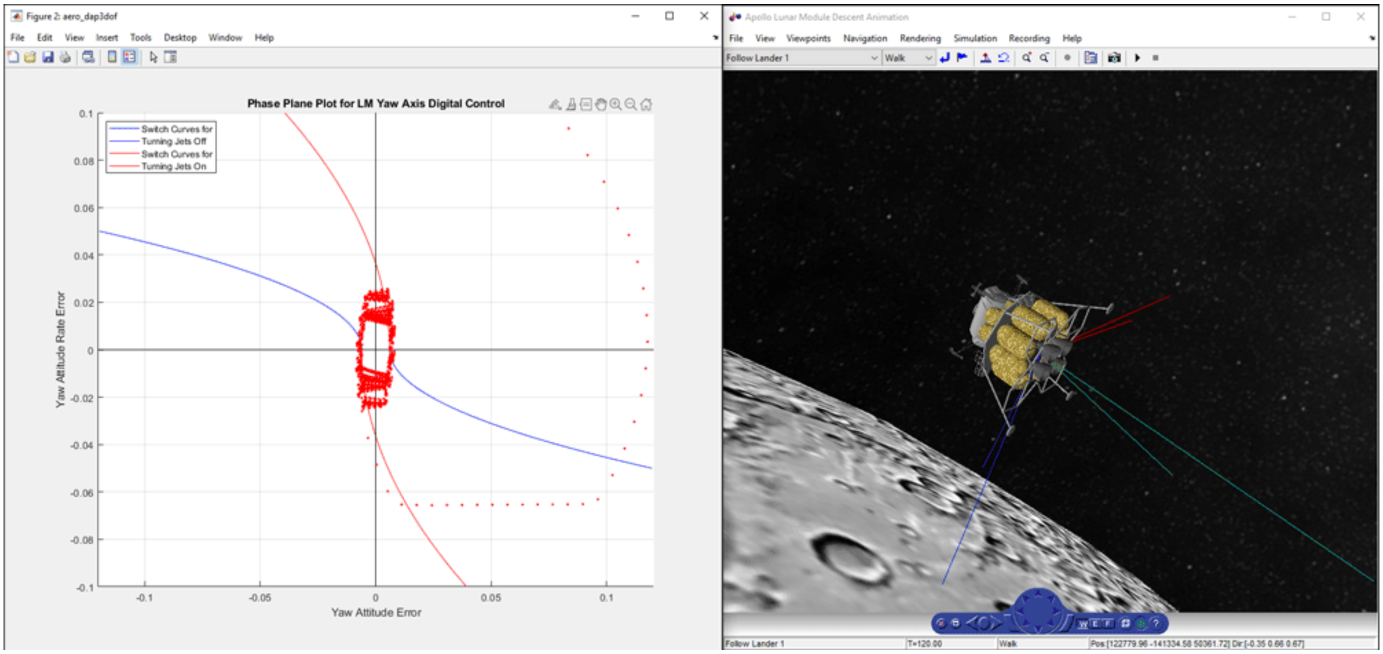
This model uses a Stateflow chart to control firing of reaction control jets positioned on the positive and negative Jet `u` and `v` axes. To locate the Stateflow chart, open the `Reaction Jet Control` subsystem, and then open the `Reaction Jet Control` subsystem within. This subsystem contains three library blocks, `Yaw Control Law`, `u Control Law` and `v Control Law`. `u Control Law` and `v Control Law` are both copies of the same library block, `Pitch Roll Control Law`. To see the Stateflow chart, open any of these three subsystems. The chart within all three library blocks is the same.



There are three main states for control: Region 1, Region 2, and Coast Region. However, the Stateflow chart implements four states: `Fire_region_1`, `Fire_region_2`, `Coast_region_1`, and `Coast_region_2`. By adding the extra coast state, the chart is able to easily flip between firing in one of the two fire jets regions and the coasting region.

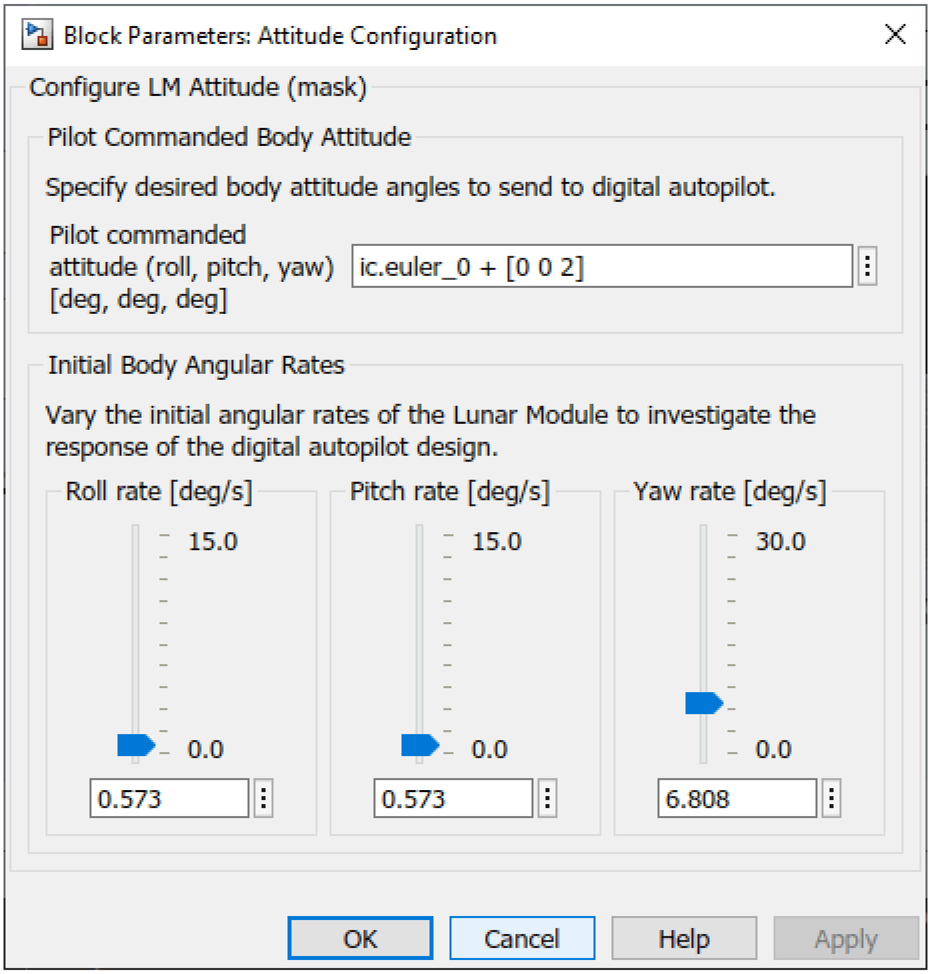


Translational and rotational dynamics of the Lunar Module are approximated in the `Lunar Module Dynamics` subsystem. Access various visualization methods of the Lunar Module states and autopilot performance in the `Visualization` area of the model, including Simulink scopes, animation with Simulink 3D Animation, and a phase plane plot.



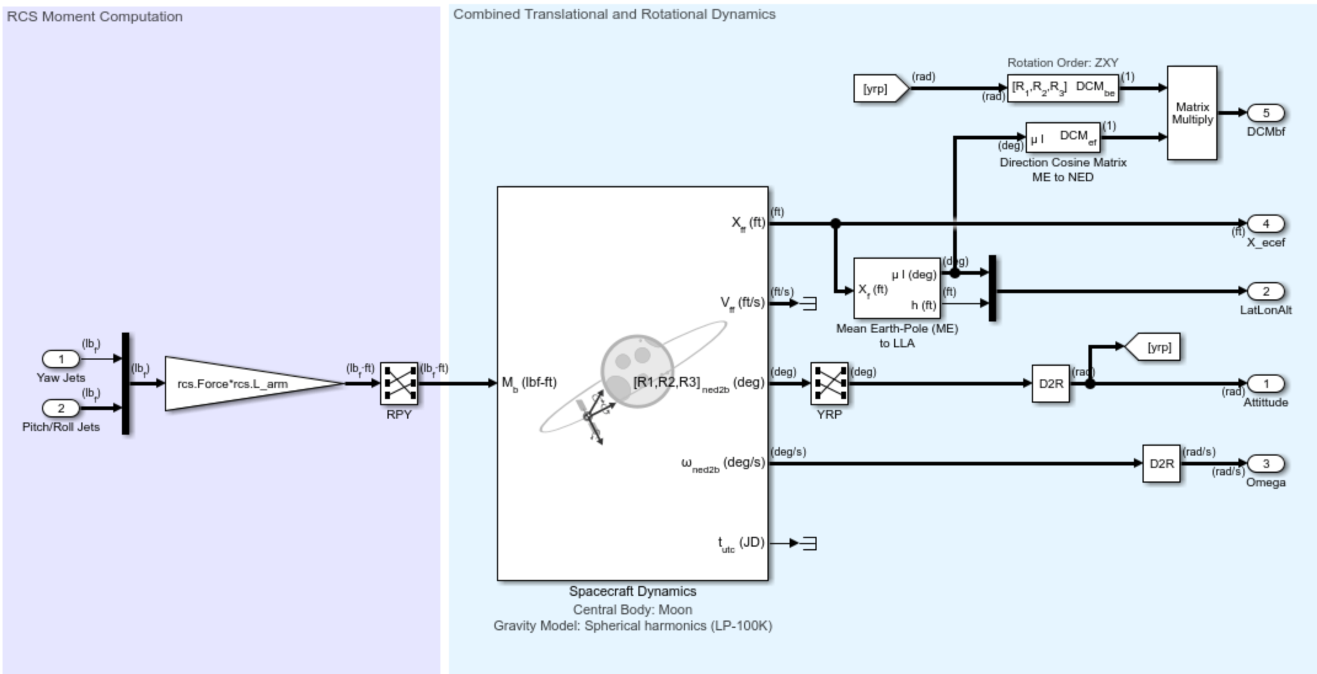
Interactive Controls

To interact with the Lunar Module model, vary autopilot settings and Lunar Module initial states in the `Commands` area. For example, to observe how the digital autopilot design handles increased initial body rates, use the slider components in `Configure LM Attitude`.



Mission Description

The LM digital autopilot has three degrees of freedom. This means that by design, the reaction jet thrusters are configured and commanded to rotate the vehicle without impacting the vehicle's orbital trajectory. Therefore, the translational dynamics in his model are approximated via orbit propagation using the Spacecraft Dynamics block from Aerospace Blockset. The block is configured to use Moon spherical harmonic gravity model LP-100K.



To demonstrate the digital autopilot design behavior, the "Descent Orbit Insertion" mission segment, just prior to the initiation of the powered descent, was selected from the *Apollo 11 Mission Report*.

NASA-S-69-3709

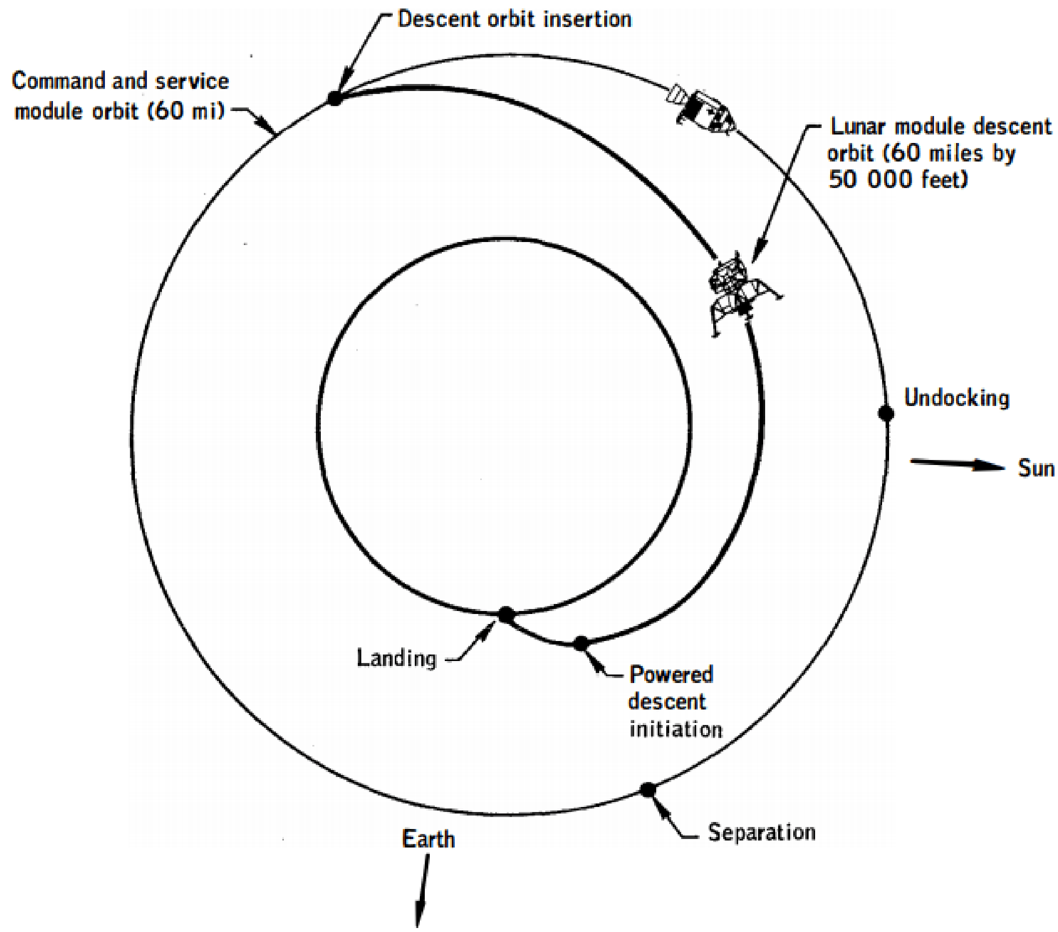


Figure 5-1. - Lunar descent orbital events.

5-19

(*Image Credit: NASA*)

The "Descent Orbit Insertion" burn began 101 hours, 36 minutes, and 14 seconds after lift-off and lasted 30 seconds. The burn set the Lunar module on a trajectory to lower its orbit from approximately 60 nautical miles to 50,000 ft over about an hour. At 50,000 ft, the Module initiated its powered descent.

Initialize the model `aero_dap3dof` with the approximate trajectory of the Lunar Module immediately after the descent orbit insertion burn.

```
mission.t_rangeZero              = datetime(1969,7,16,13,32,0); % lift-off
mission.t_descentInsertionStart = mission.t_rangeZero + hours(101) + minutes(36) + seconds(14);
mission.t_descentInsertion      = mission.t_descentInsertionStart + seconds(30);
mission.t_poweredDescentStart   = mission.t_rangeZero + hours(102) + minutes(33) + seconds(5.2);

disp(timetable([mission.t_rangeZero, mission.t_descentInsertionStart, ...
    mission.t_descentInsertion, mission.t_poweredDescentStart]', ...
    {'Range Zero (lift-off)', 'Descent Orbit Insertion (Engine ignition)', ...
    'Descent Orbit Insertion (Engine cutoff)', 'Powered Descent (Engine ignition)'}', VariableNar
```

```
         Time                               Mission Phase
    ____________________    _____________________________________________

    16-Jul-1969 13:32:00    {'Range Zero (lift-off)'                      }
    20-Jul-1969 19:08:14    {'Descent Orbit Insertion (Engine ignition)'}
    20-Jul-1969 19:08:44    {'Descent Orbit Insertion (Engine cutoff)'  }
    20-Jul-1969 20:05:05    {'Powered Descent (Engine ignition)'        }
```

The trajectory of the module at "Descent Orbit Insertion (Engine cutoff)" and "Powered Descent Initiation (Engine ignition)" is provided in the *Apollo 11 Mission Report* (Table 7-II.- Trajectory Parameters).

```
mission.Latitude_deg  = [-1.16, 1.02]';    % [deg]
mission.Longitude_deg = [-141.88, 39.39]'; % [deg]
mission.Altitude_mi   = [57.8, 6.4]';      % [nautical miles]
mission.Altitude_ft   = convlength(mission.Altitude_mi, 'naut mi', 'ft');
mission.Velocity_fps  = [5284.9, 5564.9]'; % [ft/s] (in Inertial frame)
mission.FlightPathAngle_deg = [-0.06, 0.03]'; % [deg] (measured upward from local horizontal plan
mission.HeadingAngle_deg = [-75.19 -101.23]'; % [deg] (measured East of North)
disp(table({'Range Zero (lift-off)'; 'Descent Orbit Insertion (Engine ignition)'}, ...
    mission.Latitude_deg, mission.Longitude_deg, mission.Altitude_mi, ...
    mission.Velocity_fps, mission.FlightPathAngle_deg, mission.HeadingAngle_deg, ...
    VariableNames=["Mission Phase", ...
    "Latitude (deg)", "Longitude (deg)", "Altitude (mi)", ...
    "Velocity (ft/s)", "Flight path angle (deg)", "Heading (deg)"]));
```

```
                   Mission Phase                     Latitude (deg)    Longitude (deg)    Altitu
    _____________________________________________    ______________    _______________    ______

    {'Range Zero (lift-off)'                      }       -1.16            -141.88           57
    {'Descent Orbit Insertion (Engine ignition)'}        1.02              39.39            6
```

Model Initialization

Initialize model parameters for the mission phase "Descent Orbit Insertion (Engine cutoff)" using the data defined above.

The intialization function `aero_dap3dofdata` requires information about the orientation of the Moon, which can be calculated using the Aerospace Blockset function `moonLibration`. This function requires "Ephemeris Data for Aerospace Toolbox". Use `aeroDataPackage` to install this data if it is not already installed.

```
mission.LibrationAngles_deg = moonLibration(juliandate(mission.t_descentInsertion), "405");
```

This example uses saved libration angle data corresponding with `t_descentInsertion`. Use the above command after installing the required ephemeris data.

```
mission.LibrationAngles_deg = [0.006379917345247; 0.382328074214300; 6.535718297208969];
```

Run the intialization function:

```
[moon, ic, vehicle, rcs] = aero_dap3dofdata(...
    mission.Latitude_deg(1), mission.Longitude_deg(1), mission.Altitude_ft(1), ...
    mission.Velocity_fps(1), mission.FlightPathAngle_deg(1), ...
    mission.HeadingAngle_deg(1), mission.LibrationAngles_deg)
```

```
moon = struct with fields:
    r_moon_eq: 5702428
```

```
       f_moon: 0.0012


ic = struct with fields:
       t_runtime: 120
    pos_inertial: [-3.6488e+06 -4.4381e+06 -1.9070e+06]
    vel_inertial: [4.0625e+03 -3.3792e+03 86.4867]
         euler_0: [-30 -10 -60]


vehicle = struct with fields:
    inertia_0: [3x3 double]
       mass_0: 33296


rcs = struct with fields:
     Force: 100
     L_arm: 5.5000
        DB: 0.0060
      tmin: 0.0140
     alph1: 0.0550
     alph2: 0.0039
     alph3: 0.0050
     alphu: 0.0063
     alphv: 7.8553e-04
    alphs1: 0.0055
    alphsu: 6.2855e-04
    alphsv: 7.8553e-05
    clockt: 0.0050
      delt: 0.1000
```

Closing Remarks

Building a digital autopilot was a daunting task in 1961 because there was very little industrial infrastructure for it - everything about it was in the process of being invented. Here is an excerpt from the technical article *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*:

"One reason why the [autopilot's machine code] was so complex is that the number of jets that could be used to control the rotations about the pilot axes was large. A decision was made to change the axes that the autopilot was controlling to the "jet axes" shown in `aero_dap3dof`. This change dramatically reduced the number of lines of code and made it much easier to program the autopilot in the existing computer. Without this improvement, it would have been impossible to have the autopilot use only 2000 words of storage. The lesson of this change is that when engineers are given the opportunity to code the computer with the system they are designing, they can often modify the design to greatly improve the code."

References

[1] National Aeronautics and Space Administration Manned Spacecraft Center, Mission Evaluation Team. (November 1969). *Apollo 11 Mission Report MSC-00171*. Retrieved from https://www.nasa.gov/specials/apollo50th/pdf/A11_MissionReport.pdf

[2] Richard J. Gran, MathWorks. (2019). *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*. Retrieved from https://www.mathworks.com/company/newsletters/articles/the-apollo-11-moon-landing-spacecraft-design-then-and-now.html

See Also

Zonal Harmonic Gravity Model | 6DOF ECEF (Quaternion)

External Websites

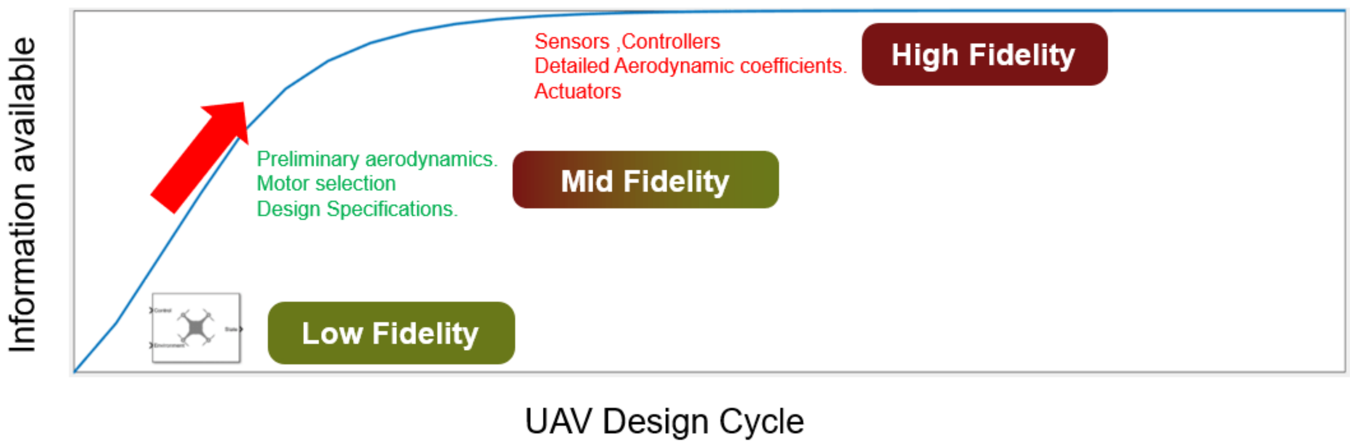
- The Apollo 11 Moon Landing: Spacecraft Design Then and Now

Transition from Low- to High-Fidelity UAV Models in Three Stages

Evolve your UAV plant model continuously to stay in sync with the latest information available.

Background

An unmanned aerial vehicle (UAV) design cycle provides incrementally better access to UAV characteristics as the design progresses. By increasing its fidelity, this information can be used to continuously evolve a plant model through a Model Based Design approach.



Towards the end of the design cycle, there is enough information to develop a high-fidelity plant. To accurately model the UAV, a high-fidelity model incorporates modeling all forces and moments, wind and environmental effects and sensors in detail. However, this level of information may be unavailable to a designer early in the design process. To build such a complex model, it can take several flight and wind tunnel tests to create enough detailed aerodynamic coefficients to compute all forces and moments that affect the UAV. These factors can potentially block guidance algorithm design until the end of the design process, when a more realistic estimate of UAV dynamics is obtained.

To concurrently design a guidance algorithm sooner, a UAV algorithm designer can start with a low-fidelity model and evolve their plant model as and when additional data becomes available.

Designing a guidance algorithm using only a low-fidelity model can also pose a risk. Without controller or aerodynamic constraints, an optimistic guidance technique can fail for a real UAV with slower aircraft dynamics.

This example highlights an alternative approach. You progress from the low-fidelity Guidance Block to a medium and then high-fidelity model by progressively adding layers of control and dynamics to the simulation. In this process, the medium-fidelity model becomes a useful tool for leveraging limited information about a plant model to tune and test guidance algorithms.

The medium-fidelity model is thus used to test a given path following an algorithm. Since the high-fidelity model is unavailable until the end of the design process, the high-fidelity model is only used later to validate our modelling approach by comparing step response and path following behavior.

Open Example and Project Files

To access the example files, click **Open Live Script** or use the `openExample` function.

```
openExample('shared_uav_aeroblks/UAVFidelityExample')
```

Open the Simulink© project provided in this example.

```
cd fidelityExample
openProject('FidelityExample.prj')
```

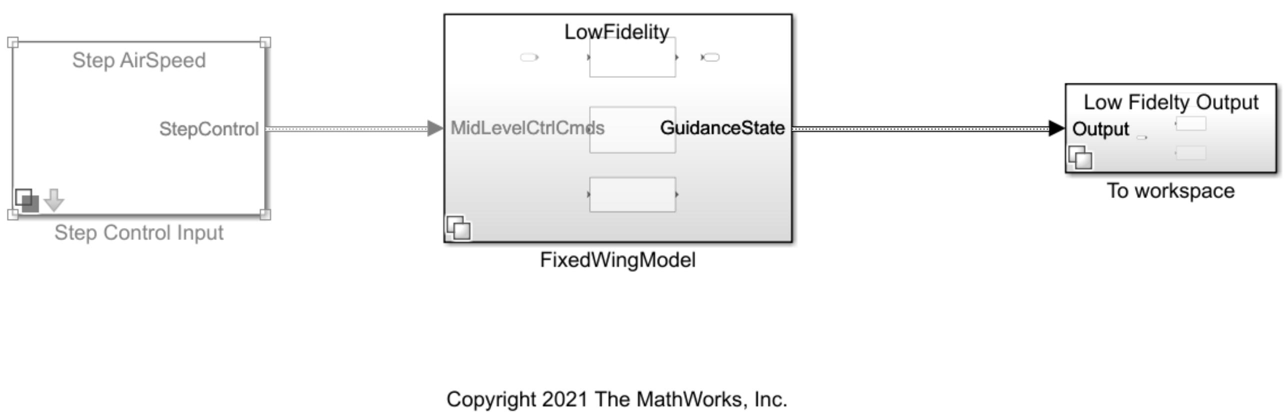
The project contains three versions of a UAV model, low-fidelity, medium-fidelity and high-fidelity with steps to study their step response and path following behaviour.

Low-Fidelity Model

Assume your UAV has the following design specifications shown in the table below. The low-fidelity variant provided in this model is tuned to achieve the desired response, but you can tune these gains for your specific requirements. The low-fidelity plant uses the UAV Guidance Block which is a reduced order model for a UAV. To run the low-fidelity variant, click the **Simulate Plant** shortcut under the **Low Fidelity** group of the project toolstrip.

Design Specification	Response Time (within 2%)	Step Change
Roll	2.5 seconds	30 degrees
Height	4.5 seconds	5 m
Airspeed	0.6 seconds	1 m/s

This shortcut sets the FidelityStage parameter to 1, configures the FidelityStepResponse model to simulate the low-fidelity model, and outputs the step response. The step response is computed for height, airspeed, and roll response.



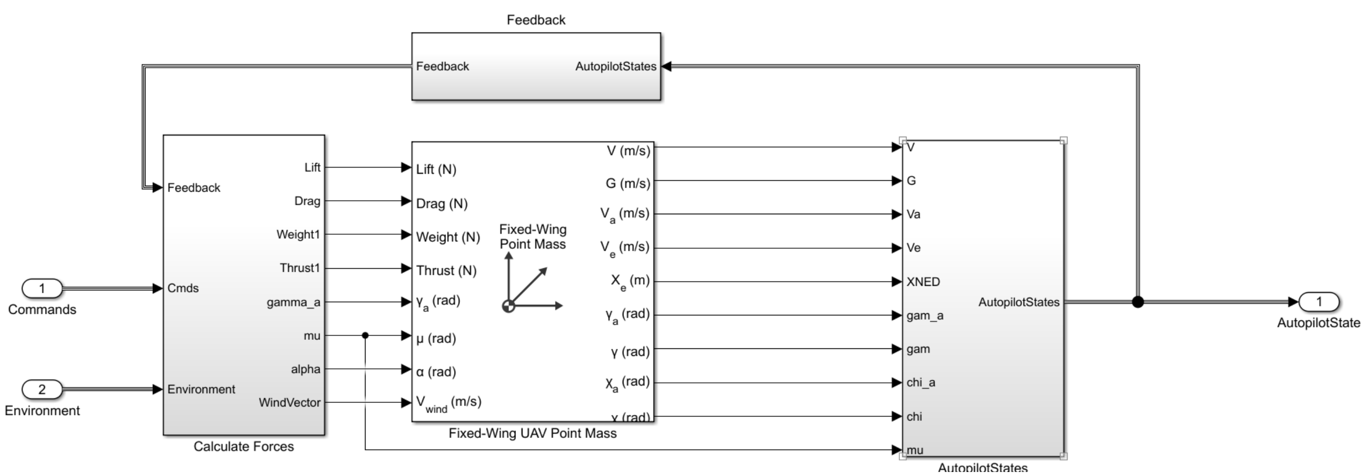
Open the UAV Fixed Wing Guidance Model block in the **FidelityStepResponse/FixedWingModel/LowFidelity** subsystem. In the Configuration tab, inspect the gains set for height, airspeed, and roll response. This guidance block integrates the controller with the dynamics of the aircraft. The low-fidelity variant gives a first estimate of how fast the UAV can realistically respond to help tune high-level planners.

Medium-Fidelity Model

As the UAV design progresses, the lift and drag coefficients become available. A motor for the aircraft is selected by the user, which defines the thrust curves. To test the validity of the guidance algorithm against this new information, the example adds this information to the plant model in this step.

To design a medium-fidelity model, the model needs only preliminary aerodynamic coefficients, thrust curves, and response time specifications. To model a medium-fidelity UAV, you can use the Fixed-Wing Point Mass Block. The block only requires lift, drag and thrust force inputs, which are much easier to approximate at an early design stage than detailed forces and moments of an aircraft. To set up the medium-fidelity variant, click the **Setup Plant** shortcut under the **Medium Fidelity** group of the project toolstrip.

Examine the Vehicle Dynamics tab in the model under **FidelityStepResponse/ FixedWingModel/Mid Fidelty/UAV Plant Dynamics/Vehicle Dynamics**.



The medium-fidelity model represents the UAV as a point mass with the primary control variables being the angle of attack and roll. This medium-fidelity plant model takes in roll, pitch, thrust as control inputs. The point mass block assumes instantaneous dynamics of roll and angle of attack. This model uses a transfer function to model roll lag based on our roll-response specification shared in the table within the previous step.

The medium-fidelity aircraft controls pitch instead of angle of attack. Since the angle of attack is an input to the point mass block, the plant model converts pitch to alpha using the following equation.

$\Theta = \gamma_a + \alpha$

Θ,γ_a and α represent pitch, flight path angle in the wind frame, and angle of attack respectively.

Unlike the low-fidelity model, the medium-fidelity model splits the autopilot from the plant dynamics. The medium-fidelity plant needs an outer-loop controller for height-pitch and airspeed-throttle control to be added. The predefined controllers provided are using standard PID-tuning loops to reach satisfactory response without overshoot. To inspect the outer-loop controller, open the `Outer_Loop_Autopilot` Simulink model.

Medium-Fidelity Step Response

The low-fidelity plant was tuned in the previous step by assuming that all response time specifications are met by the UAV. To test this assumption, use the medium-fidelity plant. The study of the step response of the improved plant is used to contrast the performance of the low-fidelity and medium-fidelity variant. To simulate the medium-fidelity step response, click the **Simulate Plant** shortcut under the **Medium Fidelity** group of the project toolstrip. The step response plots appear as figures.

Notice that the model meets the design criteria shown in the table below by achieving an air speed settling time of 0.6 seconds and a height response of 4.1 seconds. However, the height response is slower than the low-fidelity variant. This lag in response is expected due to the additional aerodynamic constraints placed on the medium-fidelity plant.

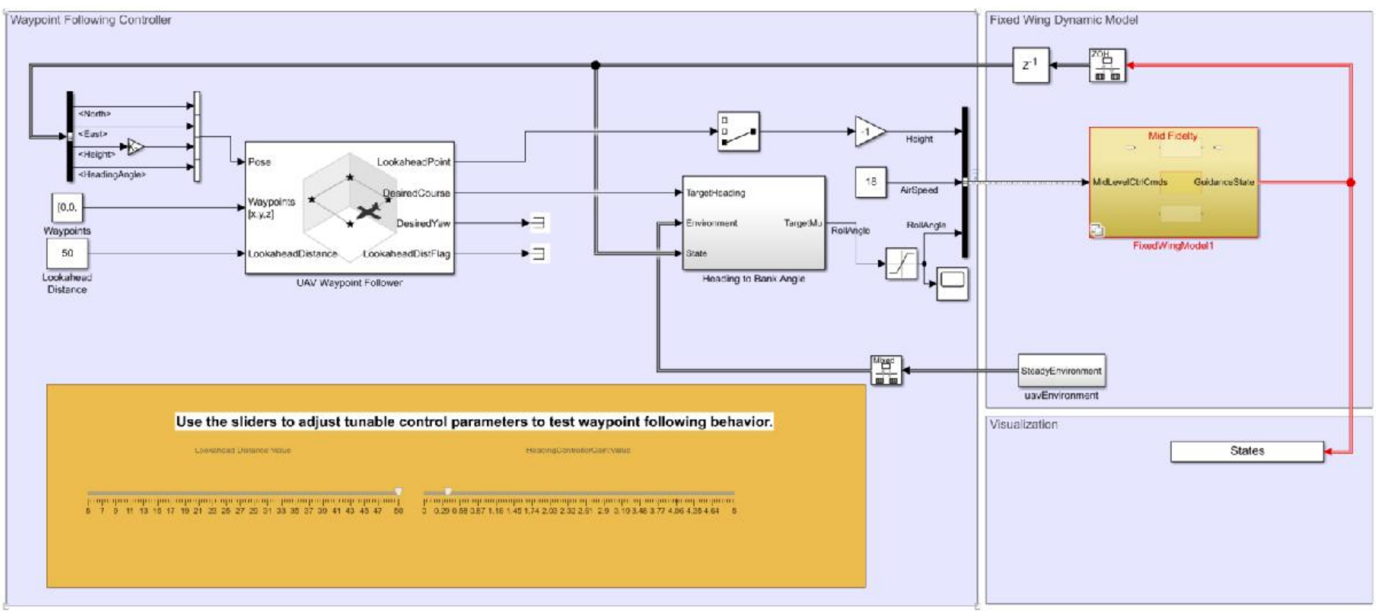
Design Specification	Response Time (within 2%)	Step Change
Roll	2.5 seconds	30 degrees
Height	4.5 seconds	5 m
Airspeed	0.6 seconds	1 m/s

Simulate Path Following Algorithm

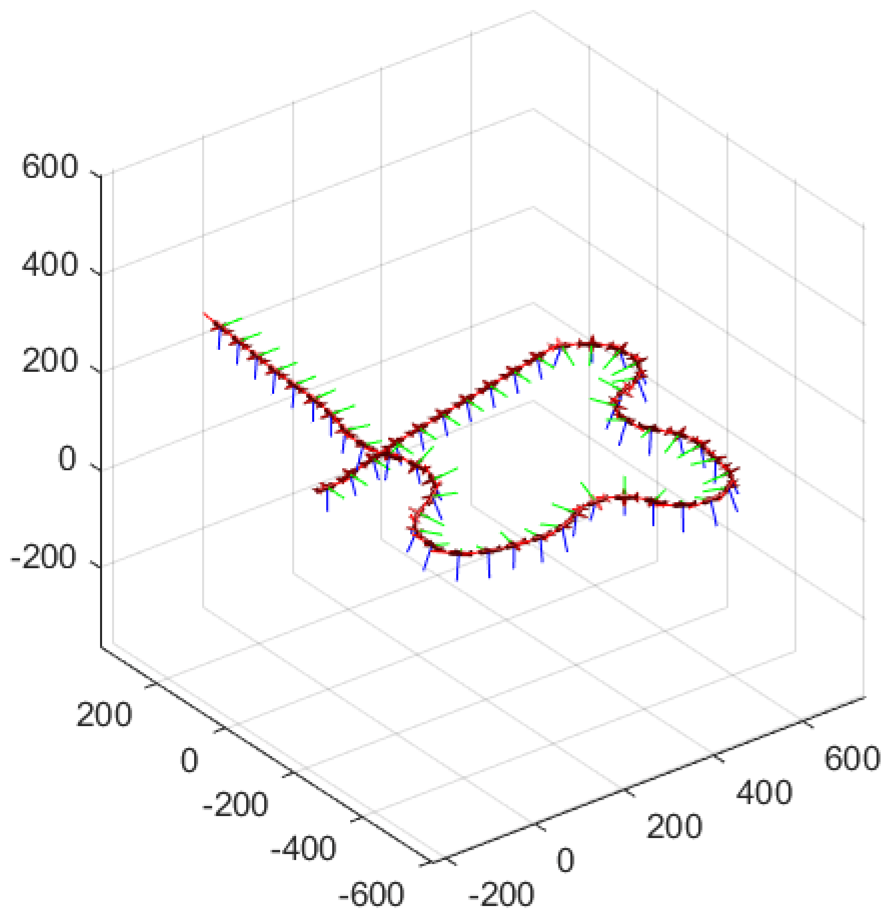
With a more accurate response from the UAV medium-fidelity model, you can now test waypoint follower or guidance algorithms to follow waypoints. For the guidance algorithm design, see the "Tuning Waypoint Follower for Fixed-Wing UAV" example.

To simulate and visualize the medium-fidelity UAV path following the model, click the **Simulate Path Follower** shortcut under the **Medium Fidelity** group of the project toolstrip.

Tune Waypoint Follower for Fixed-Wing UAV



Notice that the medium-fidelity UAV follows the desired path accurately.



High-Fidelity Step Response

The medium-fidelity model was used to test a path follower design using simple aircraft parameters available at an early design state. However, it is important to continue adding fidelity to capture UAV control response to study more complex situations. For example, the use of more detailed aerodynamics coefficients allows analysis of complex motions such as doublet maneuvers. Another example is, adding actuator dynamics lets you study the subsequent effect on inner loop controllers for attitude, which can cause destabilization. In this way, the high-fidelity plant allows refinement of control system design. In this step, to study the change in response, we look at a high-fidelity plant with these added dynamics.

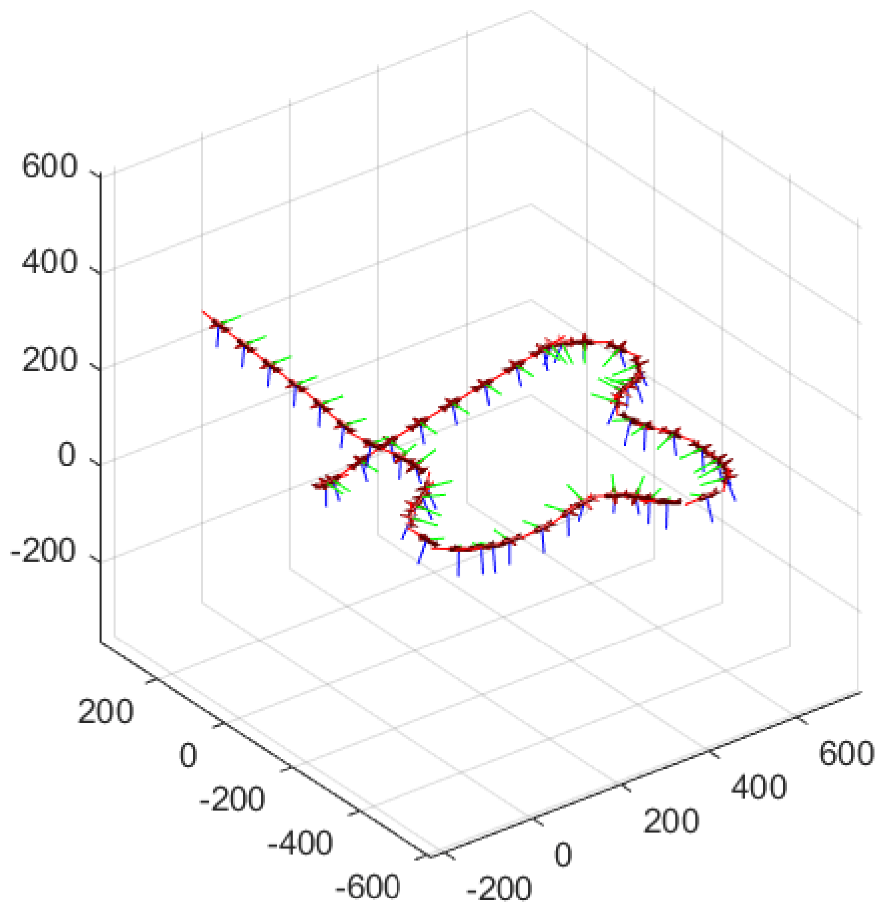
The high-fidelity plant inputs all forces and moments to a 6-DOF block, adds on-board sensors, and models actuator dynamics for the UAV. Unlike the mid-fidelity plant, the high-fidelity version does not take attitude inputs directly. Instead, an inner loop controller is added to control attitude. Additionally, a yaw compensation loop balances the non-zero sideslip. The model reuses the outer-loop controller designed for the medium-fidelity model. To validate that the medium-fidelity model provided useful intermediate information, use the response of the higher fidelity model.

To simulate and visualize the high-fidelity step response, click the **Simulate Plant** shortcut under the **High-Fidelity** group of the project toolstrip. Notice that despite added complexity, the trajectory matches well with the medium-fidelity model. Also, notice the design specifications are relatively the same for the high-fidelity stage. This similarity shows that the medium-fidelity plant modelled UAV dynamics accurately.

Design Specification	Response Time (within 2%)	Step Change
Roll	2.5 seconds	30 degrees
Height	~~4.1~~ **3.9 seconds**	5 m
Airspeed	0.6 seconds	1 m/s

Simulate Path Following Algorithm for High-Fidelity

Towards the end of the design cycle, the high-fidelity model finally becomes available. To get the final UAV path following characteristics, you can now test the guidance algorithm developed in previous steps on the high-fidelity plant. Click the **Simulate Path Follower** shortcut under the **High-Fidelity** group of the project toolstrip.



Notice that the model obtains a similar response to the medium-fidelity model using the guidance and outer-loop control parameters. This validates the guidance algorithm with a high-fidelity plant.

Conclusion

The medium-fidelity model accurately predicts the UAV dynamics making optimum use of limited information available during design. The example designs the outer loop controller and tests a waypoint follower without needing all the information in a high-fidelity plant model.

To model additional dynamics such as actuator lag, the medium-fidelity plant is flexible and can continuously evolve alongside design. The example obtains results under zero-wind conditions. In the presence of wind disturbances, the controller and path follower performance tracking might be adversely affected. To augment the autopilot controller to compensate for wind effects, leverage the atmospheric wind model in the high-fidelity plant model.

See Also

Fixed-Wing Point Mass

Lunar Mission Analysis with the Orbit Propagator Block

This example shows how to compute and visualize access intervals between the Apollo Command and Service module and a rover on the lunar surface. The module's orbit is modeled using Reference Trajectory #2 from the NASA report *Variations of the Lunar Orbital Parameters of the Apollo CSM-Module* [2]. This is a lunar orbit studied by NASA for the Apollo program. The example uses:

- Aerospace Toolbox
- Aerospace Blockset™
- Mapping Toolbox™

Define Mission Parameters and Module Initial Conditions

Specify the start date and duration for the mission. This example uses MATLAB® structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

```
mission.StartDate = datetime(1969, 9, 20, 5, 10, 12.176);
mission.Duration  = hours(2);
```

Specify Keplerian orbital elements for the Command and Service module (CSM) at the `mission.StartDate` based on Reference Trajectory #2 [2]. The criteria for the reference trajectories featured in Reference 2 are:

- The plane of the trajectory must contain a landing site vector on the Earth side of the Moon, which has a longitude of between 315 and 45 degrees and a latitude of between +5 and -5 degrees in selenographic coordinates. [2]
- The plane of the orbit must be oriented so that the lunar landing side doesn't move out of the orbital plane more than 0.5 degrees during the period of 3 to 39 hours after lunar insertion. [2]

```
csm.SemiMajorAxis  = 1894578.3;     % [m]
csm.Eccentricity   = 0.0004197061;
csm.Inclination    = 155.804726;    % [deg]
csm.RAAN           = 182.414087;    % [deg]
csm.ArgOfPeriapsis = 262.877900;    % [deg]
csm.TrueAnomaly    = 0.000824;      % [deg]
```

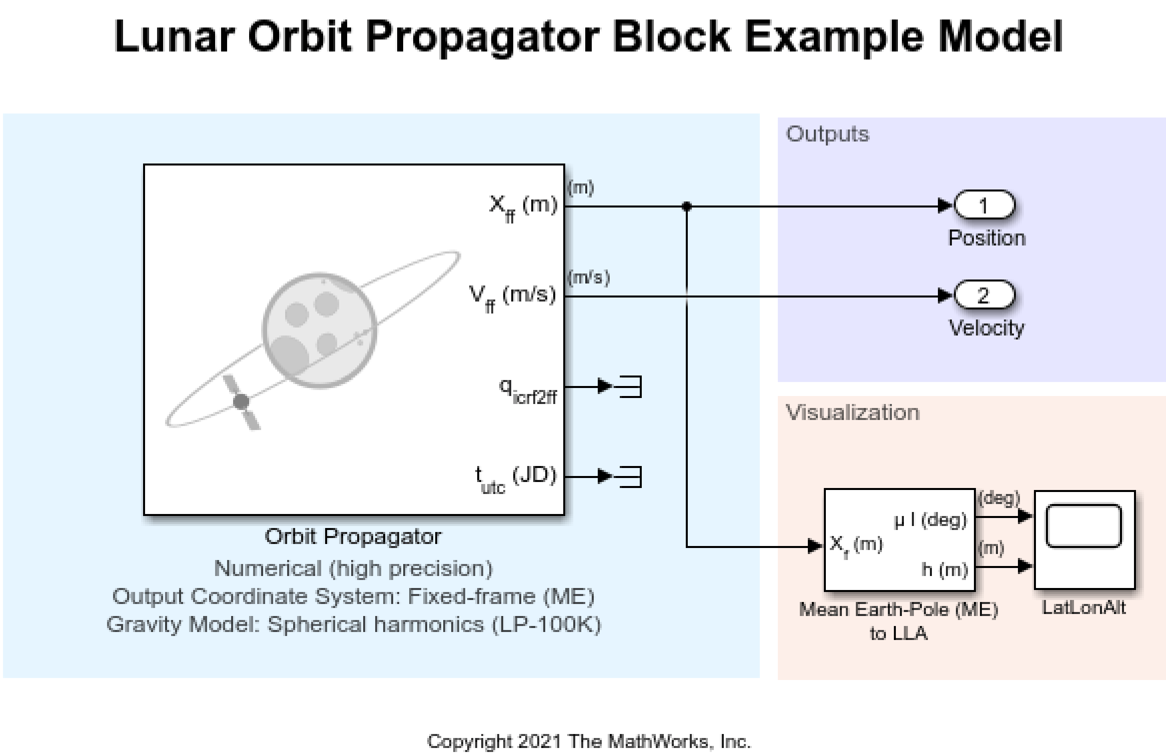
Note that the inclination angle is relative to the ICRF X-Y plane. The ICRF X-Y axis is normal to Earth's north pole. The axial tilt of Earth relative to the ecliptic is ~23.44 degrees, while the axial tilt of the Moon is ~5.145 degrees. Therefore, the axial tilt of the Moon relative to the ICRF X-Y plane varies between approximately 23.44 ± 5.145 degrees. This explains why the orbital inclination of ~155.8 degrees above satisfies the requirement to maintain a latitude of ±5 degrees in selenographic coordindates.

Specify the latitude and longitude of a rover on the lunar surface to use in the line-of-sight access analysis.

```
rover.Latitude  =   0    ;  % [deg]
rover.Longitude =  23.5  ;  % [deg]
```

Open and Configure the Model

Open the included Simulink® model. This model contains an **Orbit Propagator** block connected to output ports. The `Orbit Propagator` block supports vectorization. This allows you to model multiple satellites in a single block by specifying arrays of initial conditions in the **Block Parameters** window or using `set_param`.




```
mission.mdl = "LunarOrbitPropagatorBlockExampleModel";
open_system(mission.mdl);
```

Use a `SimulationInput` object to configure the model for our mission. `SimulationInput` objects provide the ability to configure multiple missions and run simulations with those settings without modifying the model.

```
mission.sim.in = Simulink.SimulationInput(mission.mdl);
```

Define the path to the **Orbit Propagator** block in the model.

```
csm.blk = mission.mdl + "/Orbit Propagator";
```

Load Moon properties into the base workspace.

```
moon.F = 0.0012;  % Moon ellipticity (flattening) (Ref 1)
moon.R_eq = 1737400; % [m] Lunar radius in meters (Ref 1)
moon.ReferenceEllipsoid = referenceEllipsoid("moon","meter"); % Moon reference ellipsoid
moon.Data = matfile("lunarGeographicalData.mat"); % Load moon geographical data
```

Set CSM initial conditions. To assign the Keplerian orbital element set defined in the previous section, use `setBlockParameter`.

```
mission.sim.in = mission.sim.in.setBlockParameter(...
    csm.blk, "startDate", string(juliandate(mission.StartDate)),...
    csm.blk, "stateFormatNum", "Orbital elements",...
    csm.blk, "orbitType", "Keplerian",...
    csm.blk, "semiMajorAxis", string(csm.SemiMajorAxis),...
    csm.blk, "eccentricity", string(csm.Eccentricity),...
    csm.blk, "inclination", string(csm.Inclination),...
    csm.blk, "raan", string(csm.RAAN),...
    csm.blk, "argPeriapsis", string(csm.ArgOfPeriapsis),...
    csm.blk, "trueAnomaly", string(csm.TrueAnomaly));
```

Set the position and velocity output ports of the block to use the Moon-fixed frame. The fixed-frame for the Moon is the Mean Earth/Pole Axis (ME) reference system.

```
mission.sim.in = mission.sim.in.setBlockParameter(...
    csm.blk, "centralBody", "Moon",...
    csm.blk, "outportFrame", "Fixed-frame");
```

Configure the propagator.

```
mission.sim.in = mission.sim.in.setBlockParameter(...
    csm.blk, "propagator", "Numerical (high precision)",...
    csm.blk, "gravityModel", "Spherical Harmonics",...
    csm.blk, "moonSH", "LP-100K",... % moon spherical harmonic potential model
    csm.blk, "shDegree", "100",... % Spherical harmonic model degree and order
    csm.blk, "useMoonLib", "off");
```

Apply model-level solver settings using setModelParameter. For best performance and accuracy when using a numerical propagator, use a variable-step solver.

```
mission.sim.in = mission.sim.in.setModelParameter(...
    SolverType="Variable-step",...
    SolverName="VariableStepAuto",...
    RelTol="1e-6",...
    AbsTol="1e-7",...
    StopTime=string(seconds(mission.Duration)));
```

Save model output port data as a dataset of timetable objects.

```
mission.sim.in = mission.sim.in.setModelParameter(...
    SaveOutput="on",...
    OutputSaveName="yout",...
    SaveFormat="Dataset",...
    DatasetSignalFormat="timetable");
```

Run the Model and Collect CSM Ephemerides

Simulate the model using the `SimulationInput` object defined above. In this example, the **Orbit Propagator** block is set to output position and velocity states in the Moon-centered fixed coordinate frame.

```
mission.sim.out = sim(mission.sim.in);
```

Extract the position and velocity data from the model output data structure.

```
csm.TimetablePos = mission.sim.out.yout{1}.Values;
csm.TimetableVel = mission.sim.out.yout{2}.Values;
```

Set the start date of the mission in the `timetable` object.

```
csm.TimetablePos.Properties.StartTime = mission.StartDate;
```

Process Simulation Data

Compute latitude, longitude, and altitude using lunar equatorial radius and flattening. Values are displayed in degrees and meters.

```
csm.MEPos = [csm.TimetablePos.Data(:,1) ...
    csm.TimetablePos.Data(:,2) csm.TimetablePos.Data(:,3)];
lla = ecef2lla(csm.MEPos, moon.F, moon.R_eq);
csm.LLA = timetable(csm.TimetablePos.Time, ...
    lla(:,1), lla(:,2), lla(:,3), ...
    VariableNames=["Lat", "Lon", "Alt"]);
clear lla;
disp(csm.LLA);
```

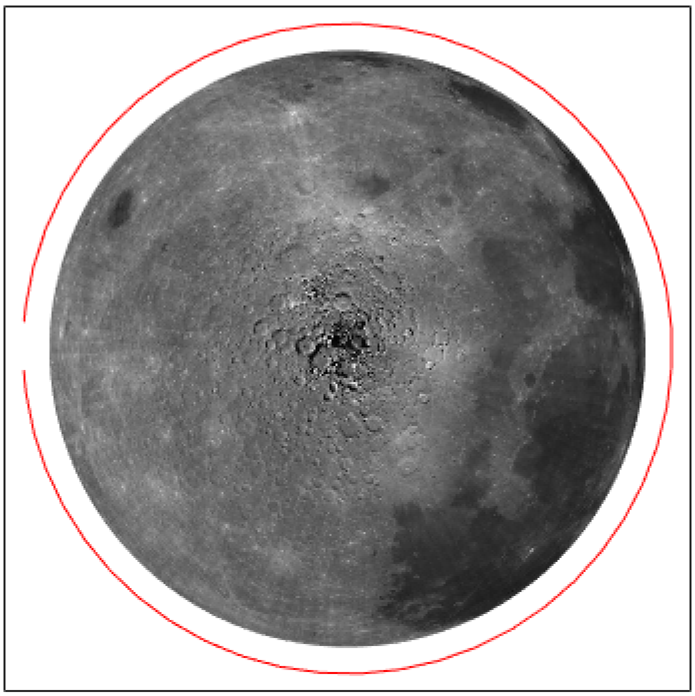
```
            Time               Lat          Lon          Alt    
    ____________________    _________    _______    __________

    20-Sep-1969 05:10:12      -2.3072     175.32    1.5639e+05
    20-Sep-1969 05:10:22      -2.3039     174.83    1.5639e+05
    20-Sep-1969 05:11:12      -2.2846     172.39    1.5639e+05
    20-Sep-1969 05:13:36      -2.2061     165.35    1.5639e+05
    20-Sep-1969 05:16:00      -2.0947     158.31     1.564e+05
    20-Sep-1969 05:18:24       -1.952     151.27    1.5641e+05
    20-Sep-1969 05:20:48      -1.7804     144.24    1.5641e+05
    20-Sep-1969 05:23:12      -1.5824     137.21    1.5642e+05
    20-Sep-1969 05:25:36      -1.3608     130.17    1.5641e+05
    20-Sep-1969 05:28:00       -1.119     123.14    1.5641e+05
    20-Sep-1969 05:30:24     -0.86057     116.11     1.564e+05
    20-Sep-1969 05:32:48     -0.58934     109.09     1.564e+05
    20-Sep-1969 05:35:12     -0.30942     102.06    1.5639e+05
    20-Sep-1969 05:37:36    -0.025001     95.032    1.5639e+05
    20-Sep-1969 05:40:00      0.25967     88.006     1.564e+05
    20-Sep-1969 05:42:24      0.54034     80.978     1.564e+05
    20-Sep-1969 05:44:48      0.81284     73.951    1.5641e+05
    20-Sep-1969 05:47:12       1.0732     66.923    1.5642e+05
    20-Sep-1969 05:49:36       1.3175     59.893    1.5643e+05
    20-Sep-1969 05:52:00       1.5422     52.863    1.5646e+05
    20-Sep-1969 05:54:24       1.7439     45.831    1.5649e+05
    20-Sep-1969 05:56:48       1.9194     38.797    1.5652e+05
    20-Sep-1969 05:59:12       2.0662     31.763    1.5656e+05
    20-Sep-1969 06:01:36       2.1821     24.728     1.566e+05
    20-Sep-1969 06:04:00       2.2652     17.691    1.5664e+05
    20-Sep-1969 06:06:24       2.3145     10.655    1.5668e+05
    20-Sep-1969 06:08:48       2.3291     3.6183    1.5673e+05
    20-Sep-1969 06:11:12        2.309     -3.418    1.5676e+05
    20-Sep-1969 06:13:36       2.2544    -10.454    1.5679e+05
    20-Sep-1969 06:16:00       2.1663    -17.489    1.5682e+05
    20-Sep-1969 06:18:24        2.046    -24.522    1.5683e+05
    20-Sep-1969 06:20:48       1.8953    -31.554    1.5685e+05
    20-Sep-1969 06:23:12       1.7163    -38.585    1.5686e+05
    20-Sep-1969 06:25:36       1.5116    -45.614    1.5686e+05
    20-Sep-1969 06:28:00       1.2844    -52.642    1.5686e+05
    20-Sep-1969 06:30:24       1.0381    -59.668    1.5686e+05
    20-Sep-1969 06:32:48      0.77625    -66.693    1.5685e+05
    20-Sep-1969 06:35:12      0.50273    -73.718    1.5684e+05
    20-Sep-1969 06:37:36      0.22159    -80.741    1.5683e+05
```

```
20-Sep-1969 06:40:00    -0.062926    -87.765    1.5682e+05
20-Sep-1969 06:42:24     -0.34651    -94.789     1.568e+05
20-Sep-1969 06:44:48     -0.62489    -101.81    1.5677e+05
20-Sep-1969 06:47:12     -0.89393    -108.84    1.5673e+05
20-Sep-1969 06:49:36      -1.1497    -115.87    1.5669e+05
20-Sep-1969 06:52:00      -1.3884    -122.89    1.5664e+05
20-Sep-1969 06:54:24      -1.6064    -129.92     1.566e+05
20-Sep-1969 06:56:48      -1.8006    -136.96    1.5656e+05
20-Sep-1969 06:59:12      -1.9679    -143.99    1.5652e+05
20-Sep-1969 07:01:36      -2.1058    -151.03    1.5647e+05
20-Sep-1969 07:04:00       -2.212    -158.06    1.5641e+05
20-Sep-1969 07:06:24      -2.2849     -165.1    1.5635e+05
20-Sep-1969 07:08:48      -2.3235    -172.14     1.563e+05
20-Sep-1969 07:10:12      -2.3299    -176.25    1.5626e+05
```

Results

Display CSM Trajectories over the 3-D Moon

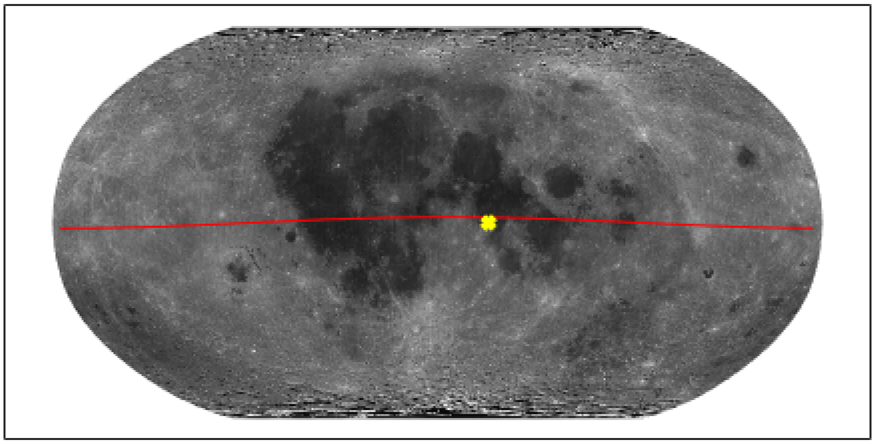
```
figure; axis off; colormap gray; view(-5,23);
axesm("globe","Geoid", moon.ReferenceEllipsoid);
geoshow(moon.Data.moonalb20c, moon.Data.moonalb20cR, DisplayType="texturemap");
plot3(csm.MEPos(:,1), csm.MEPos(:,2), csm.MEPos(:,3),"r");
```



Display 2-D Projection of CSM Ground Trace and Rover Position

```
figure; colormap gray;
axesm(MapProjection="robinson");
```

```
geoshow(moon.Data.moonalb20c, moon.Data.moonalb20cR, DisplayType="texturemap");
plotm(csm.LLA.Lat, csm.LLA.Lon, Color="r");
plotm(rover.Latitude, rover.Longitude, "xy", LineWidth=3);
```



Display CSM Field of View at Time of Interest

Define a time of interest (TOI) to anayze. For this example, we select the 30th sample in the dataset.

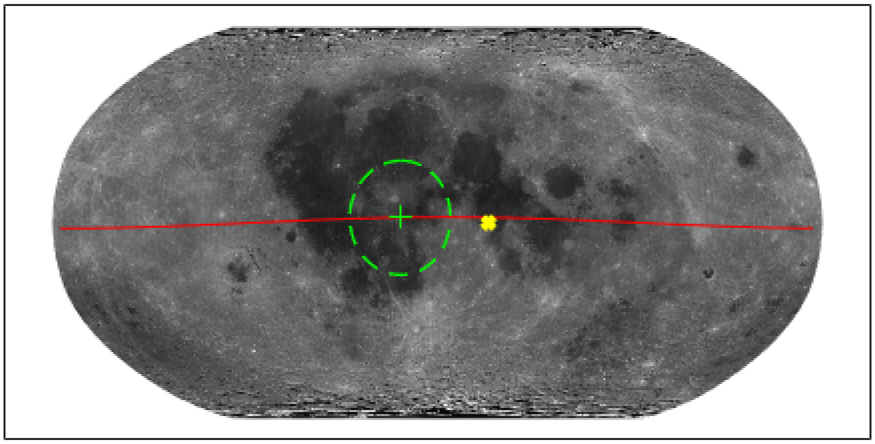
```
csm.TOI.LLA = csm.LLA(30,:);
```

Calculate angular radius of orbiter line-of-sight (LOS) field of view (FOV) measured from the Moon center.

```
csm.TOI.FOV.Lambda0 = acosd(moon.R_eq / (moon.R_eq + csm.TOI.LLA.Alt)); % [deg]
[csm.TOI.FOV.Lats, csm.TOI.FOV.Lons] = ...
    scircle1(csm.TOI.LLA.Lat, csm.TOI.LLA.Lon, csm.TOI.FOV.Lambda0);
```

Plot TOI. The location of the CSM is indicated by a green cross, LOS field of view is indicated by dashed circle.

```
figure; colormap gray;
axesm(MapProjection="robinson");
geoshow(moon.Data.moonalb20c, moon.Data.moonalb20cR, DisplayType="texturemap");
plotm(csm.TOI.FOV.Lats, csm.TOI.FOV.Lons, "g--", LineWidth=1); % CSM visibility projected onto th
plotm(csm.LLA.Lat, csm.LLA.Lon, Color="r");                    % CSM ground trace
plotm(csm.TOI.LLA.Lat, csm.TOI.LLA.Lon, "g+", MarkerSize=8);   % sub-CSM point
plotm(rover.Latitude, rover.Longitude, "xy", LineWidth=3);
```



Display CSM Line-of-Sight Visibility from Rover

Estimate access intervals by assuming the Moon is spherical.

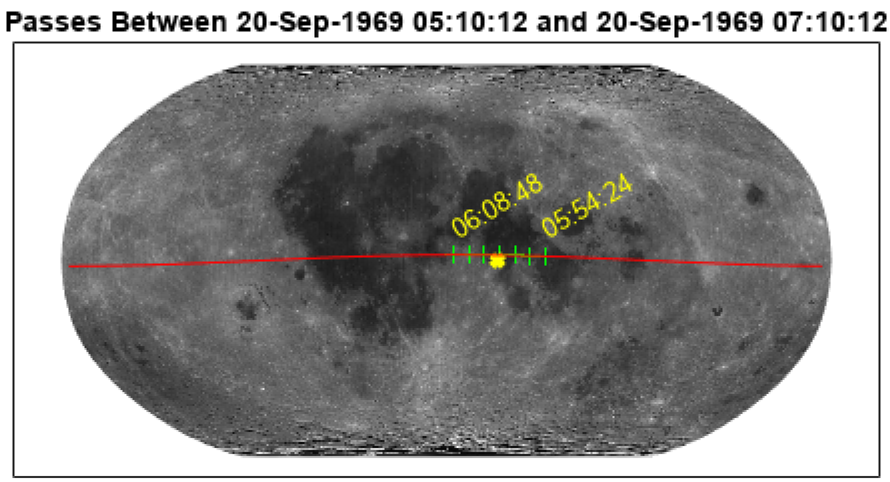
```
lambda_all  = acosd(moon.R_eq ./ (moon.R_eq + csm.LLA.Alt)); % [deg] angular radius of CSM FOV me
d = distance(csm.LLA.Lat, csm.LLA.Lon, ...
    rover.Latitude, rover.Longitude);                        % [deg] angular distance between sub
rover.Access.InView = csm.LLA(lambda_all - d > 0,:);         % timetable containing the in view d
rover.Access.InView.Time.Format = "HH:mm:ss";
clear lambda_all d;
```

Plot access intervals between the orbiting module and rover.

```
if height(rover.Access.InView) ~= 0
    % Look for breaks in the timestamps to identify pass starts
    rover.Access.StartIdx = [1, find(diff(rover.Access.InView.Time) > minutes(5))];
    rover.Access.StartTime = rover.Access.InView.Time(rover.Access.StartIdx);
    rover.Access.StopIdx = [rover.Access.StartIdx(2:end)-1, height(rover.Access.InView)];
    rover.Access.StopTime = rover.Access.InView.Time(rover.Access.StopIdx);
    rover.Access.Duration = rover.Access.StopTime - rover.Access.StartTime;
    % Show pass intervals in table
    rover.Access.IntervalTable = table(rover.Access.StartTime, rover.Access.StopTime, rover.Acces
        VariableNames=["Pass Start", "Pass End", "Duration"]);
    disp(rover.Access.IntervalTable);
    disp(' ');
    % Set up figure window/plot
    figure; colormap gray;
    axesm(MapProjection="robinson")
```

```
    geoshow(moon.Data.moonalb20c, moon.Data.moonalb20cR, DisplayType="texturemap")
    title(join(["Passes Between", string(csm.LLA.Time(1)), ...
        "and", string(csm.LLA.Time(end))]));
    % Plot inView, rover, and CSM orbit
    plotm(rover.Access.InView.Lat, rover.Access.InView.Lon, "+g");
    plotm(rover.Latitude, rover.Longitude, "xy", LineWidth=3);
    plotm(csm.LLA.Lat, csm.LLA.Lon, Color="r");
    % Plot pass inteval
    rover.Access.EdgeIndices = rover.Access.InView(sort([rover.Access.StartIdx rover.Access.StopI
    for j = 1 : height(rover.Access.EdgeIndices)
        textm(rover.Access.EdgeIndices.Lat(j) + 10, ...
            rover.Access.EdgeIndices.Lon(j), ...
            string(rover.Access.EdgeIndices.Time(j)), Color="y", Rotation=30);
    end
else
    disp("The CSM is not visible from the rover during the defined mission time.")
end
```

```
    Pass Start    Pass End    Duration
    __________    ________    ________

     05:54:24     06:08:48    00:14:24
```



References

[1] Williams, Dr. David R. “Moon Fact Sheet”, *Planetary Fact Sheets*, NSSDCA, NASA Goddard Space Flight Center, 13 January 2020, https://nssdc.gsfc.nasa.gov/planetary/factsheet/moonfact.html.

[2] Timer, T.P. (NASA Mission Analysis Office) "Variations of the Lunar Orbital Parameters of the Apollo CSM-Module", NASA TM X-55460. Greenbelt, Maryland: Goddard Space Flight Center, February 1966.

See Also

Orbit Propagator

Analyzing Spacecraft Attitude Profiles with Satellite Scenario

This example shows how to propagate the orbit and attitude states of a satellite in Simulink® and visualize the computed trajectory and attitude profile in a satellite scenario. It uses:

- Aerospace Blockset™ **Spacecraft Dynamics** block
- Aerospace Blockset **Attitude Profile** block
- Aerospace Toolbox **`satelliteScenario`** object

The **Spacecraft Dynamics** block models translational and rotational dynamics of spacecraft using numerical integration. It computes the position, velocity, attitude, and angular velocity of one or more spacecraft over time. For the most accurate results, use a variable step solver with low tolerance settings (less than 1e-8). Depending on your mission requirements, you can increase speed by using larger tolerances. Doing so might impact the accuracy of the solution.

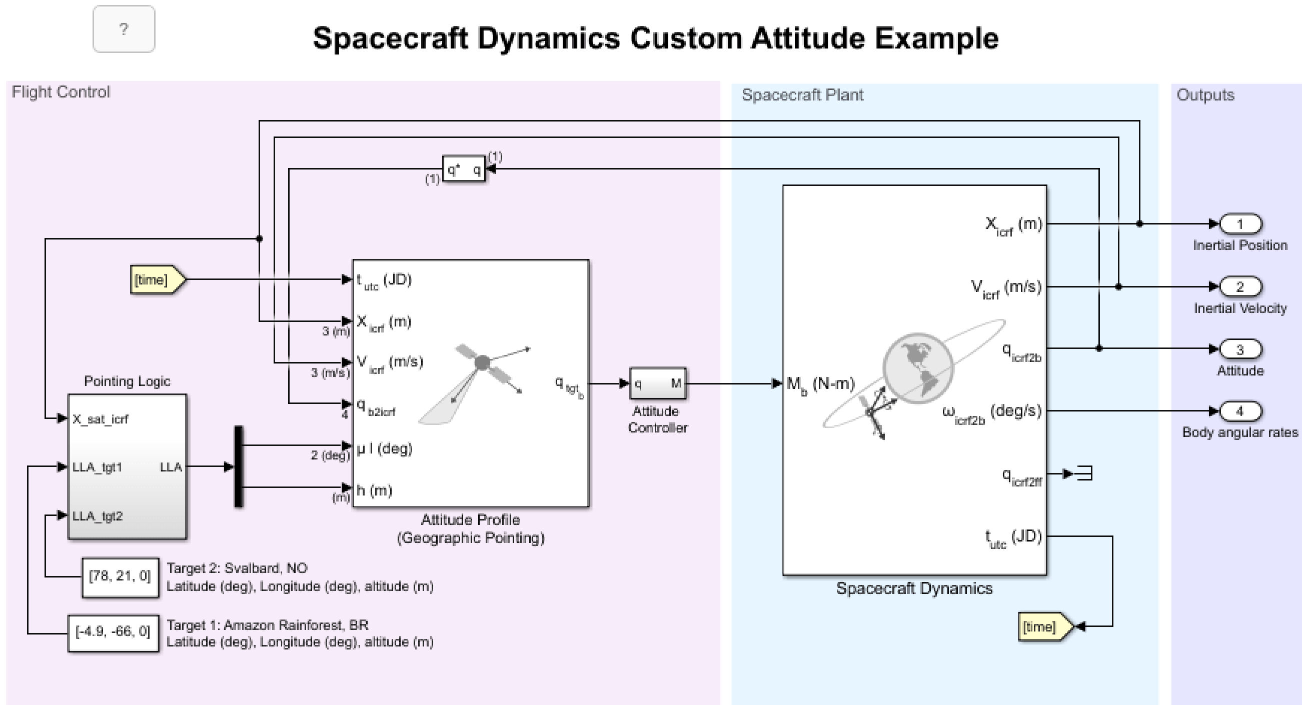
The **Attitude Profile** block returns the shortest quaternion rotation that aligns the satellite's provided alignment axis with the specified target. In this example, the satellite points towards the nadir at the beginning of the mission, then slews to align with Target 1, points back at the nadir, then slews to point at Target 2. Both targets are provided as geographic locations.

The Aerospace Toolbox **`satelliteScenario`** object lets you load previously generated, time-stamped ephemeris and attitude data into a scenario as timeseries or timetable objects. Data is interpolated in the scenario object to align with the scenario time steps, allowing you to incorporate data generated in a Simulink model into either a new or existing **`satelliteScenario`** object. In this example, the satellite orbit and attitude states are computed with the Spacecraft Dynamics block, then this data is used to add a satellite to a new **`satelliteScenario`** object for access analysis.

Open the Example Model

The example model is configured to perform an Earth Observation mission during which a satellite performs a flyover of a region of the Amazon Rainforest to capture images of, and track deforestation trends in, the area. The satellite points at the nadir when not actively imaging or downlinking to the ground station in Svalbard, NO.

```
mission.mdl = "SpacecraftDynamicsCustomAttitudeExampleModel";
open_system(mission.mdl);
```



Define Mission Parameters and Satellite Initial Conditions

Specify a start date and duration for the mission. This example uses MATLAB® structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

```
mission.StartDate = datetime(2021,1,1,12,0,0);
mission.Duration = hours(1.5);
```

Set Satellite Properties on Spacecraft Dynamics Block

Specify initial orbital elements for the satellite.

```
mission.Satellite.blk = mission.mdl + "/Spacecraft Dynamics";
mission.Satellite.SemiMajorAxis  = 7.2e6; % meters
mission.Satellite.Eccentricity   = .05;
mission.Satellite.Inclination    = 70;    % deg
mission.Satellite.ArgOfPeriapsis = 0;     % deg
mission.Satellite.RAAN           = 215;   % deg
mission.Satellite.TrueAnomaly    = 200;   % deg
```

Specify an initial attitude state for the satellite.

```
mission.Satellite.q0 = [0.1509 0.4868 0.3031 -0.8052];
mission.Satellite.pqr = [0, 0, 0]; % deg/s
```

Configure the Spacecraft Dynamics block with the provided initial conditions and desired propagation settings. These values can also be set from the Property Inspector in Simulink.

```
set_param(mission.Satellite.blk, ...
    "startDate",      string(juliandate(mission.StartDate)), ...
```

```
    "stateFormatNum", "Orbital elements", ...
    "orbitType",      "Keplerian", ...
    "semiMajorAxis",  string(mission.Satellite.SemiMajorAxis), ...
    "eccentricity",   string(mission.Satellite.Eccentricity), ...
    "inclination",    string(mission.Satellite.Inclination), ...
    "raan",           string(mission.Satellite.RAAN), ...
    "argPeriapsis",   string(mission.Satellite.ArgOfPeriapsis), ...
    "trueAnomaly",    string(mission.Satellite.TrueAnomaly));
set_param(mission.Satellite.blk, ...
    "attitudeFormat", "Quaternion", ...
    "attitudeFrame",  "ICRF", ...
    "attitude",       mat2str(mission.Satellite.q0), ...
    "attitudeRate",   mat2str(mission.Satellite.pqr));
```

Use the EGM2008 spherical harmonic gravity model for orbit propagation.

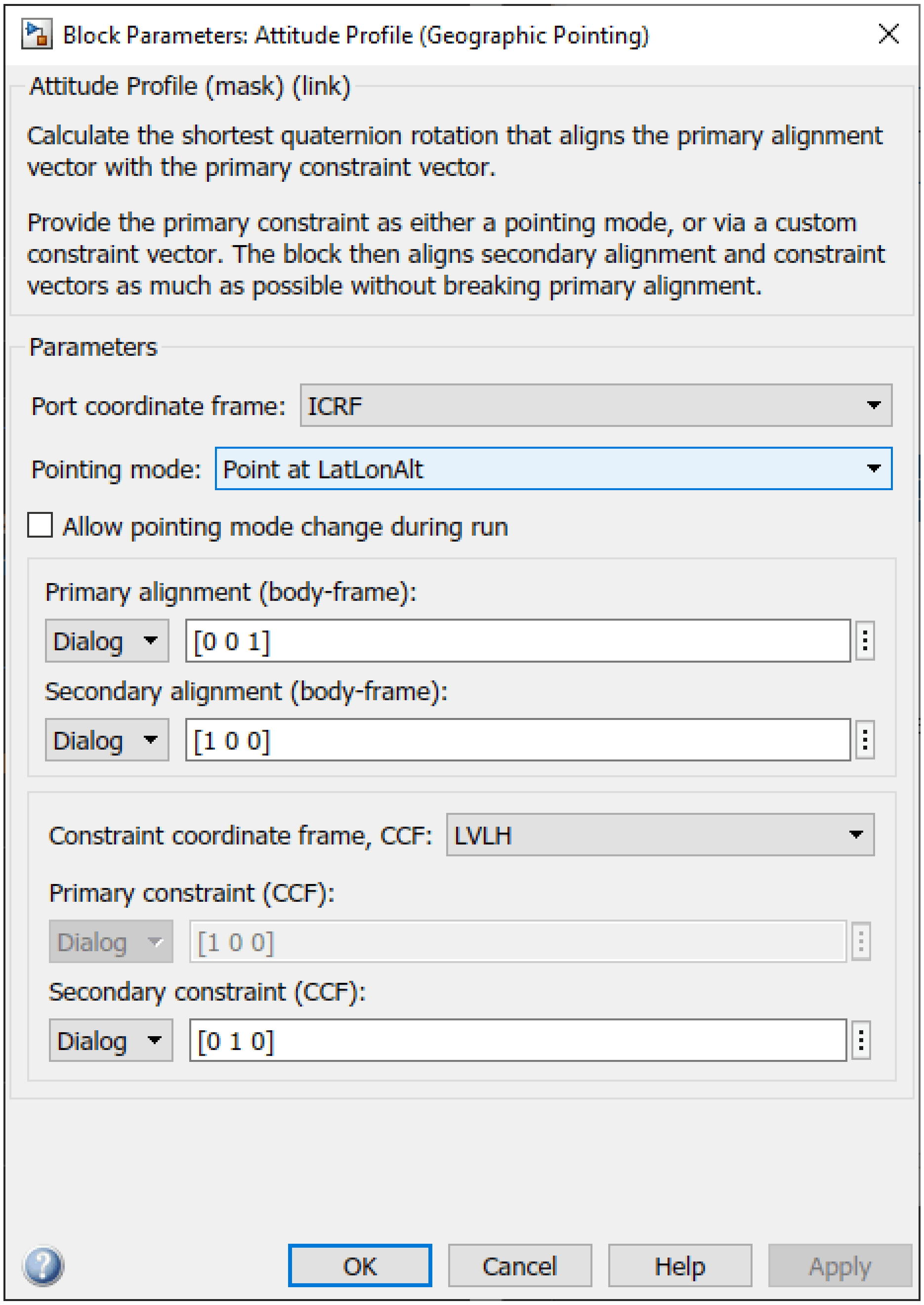
```
set_param(mission.Satellite.blk, ...
    "gravityModel", "Spherical Harmonics", ...
    "earthSH",      "EGM2008", ... % Earth spherical harmonic potential model
    "shDegree",     "120", ... % Spherical harmonic model degree and order
    "useEOPs",      "on", ... % Use EOP's in ECI to ECEF transformations
    "eopFile",      "aeroiersdata.mat"); % EOP data file
```

Gravity gradient torque contributions can be included in attitude dynamics calculations.

```
set_param(mission.Satellite.blk, "useGravGrad", "on");
```

Configure Attitude Profile Block for Target Pointing

The Attitude Profile block targets two ground locations, first a location in the Amazon Rainforest of Brazil for observation of deforestation, and second for down-linking image data to a ground station in Svalbard, NO. The block is preconfigured in the model as shown below.



The "Point at LatLonAlt" option is selected for the **Pointing mode** parameter. The *z*-axis is used as the satellite's primary alignment vector. This means that the satellite Body *z*-axis points towards the geographic coordinates passed into the block throughout the simulation. The *y*-Axis of the LVLH frame, which points along-track in the direction of travel, is defined as the secondary constraint vector. The satellite Body *x*-axis is specified as the secondary alignment vector. This keeps our satellite pointed forward throughout the mission as much as possible without disrupting primary alignment.

Set Up Simulink Model to Produce Desired Output

Apply model-level solver setting using `set_param`. For best performance and accuracy, use a variable-step solver. Set the max step size to a value that results in output data without large time gaps.

```
set_param(mission.mdl, ...
    "SolverType", "Variable-step", ...
    "SolverName", "VariableStepAuto", ...
    "RelTol",     "0.5e-5", ...
    "AbsTol",     "1e-5", ...
    "MaxStep",    "5", ...
    "MinStep",    "auto", ...
    "StopTime",   string(seconds(mission.Duration)));
```

Save model output port data as a `dataset` of `timetable` objects.

```
set_param(mission.mdl, ...
    "SaveOutput", "on", ...
    "OutputSaveName", "yout", ...
    "SaveFormat", "Dataset", ...
    "DatasetSignalFormat", "timetable");
```

Run the Model and Collect Satellite Ephemeris and Attitude Profile

Simulate the model. In this example, the **Spacecraft Dynamics** block outputs position and velocity states in the inertial (ICRF/GCRF) coordinate frame.

```
mission.SimOutput = sim(mission.mdl);
```

Create and Visualize the Satellite Scenario

For the analysis, create a satellite scenario object. Specify a timestep of 1 minute.

```
scenario = satelliteScenario(mission.StartDate, ...
    mission.StartDate + mission.Duration, 60);
```

Add the two targets as ground stations in Brazil and Svalbard.

```
gsNO = groundStation(scenario, 78, 21, Name="Svalbard, NO")
```

```
gsNO = 
  GroundStation with properties:
                 Name:  Svalbard, NO
                   ID:  1
             Latitude:  78 degrees
            Longitude:  21 degrees
             Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
       CoordinateAxes:  [1x1 matlabshared.satellitescenario.CoordinateAxes]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
```

```
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

```
gsAmazon = groundStation(scenario, -4.9, -66, Name="Amazon Rainforest")
```

```
gsAmazon = 
  GroundStation with properties:
                 Name:  Amazon Rainforest
                   ID:  2
             Latitude:  -4.9 degrees
            Longitude:  -66 degrees
             Altitude:  0 meters
    MinElevationAngle:  0 degrees
       ConicalSensors:  [1x0 matlabshared.satellitescenario.ConicalSensor]
              Gimbals:  [1x0 matlabshared.satellitescenario.Gimbal]
         Transmitters:  [1x0 satcom.satellitescenario.Transmitter]
            Receivers:  [1x0 satcom.satellitescenario.Receiver]
             Accesses:  [1x0 matlabshared.satellitescenario.Access]
       CoordinateAxes:  [1x1 matlabshared.satellitescenario.CoordinateAxes]
          MarkerColor:  [1 0.4118 0.1608]
           MarkerSize:  6
            ShowLabel:  true
       LabelFontColor:  [1 1 1]
        LabelFontSize:  15
```

Add the observation satellite to the scenario. Update the position `timetable` data in the `SimOutput` object to remove excess data points.

```
mission.Satellite.Ephemeris = retime(mission.SimOutput.yout{1}.Values, ...
    seconds(uniquetol(mission.SimOutput.tout, .0001)));
sat = satellite(scenario, mission.Satellite.Ephemeris, ...
    "CoordinateFrame", "inertial", "Name", "ObservationSat");
```

Add a conical sensor to the satellite, with a 35 deg half angle to represent the onboard camera. Enable field of view visualization in the scenario viewer. To assist in visualization, the sensor is mounted 10m from the satellite, in the $+z$ direction.

```
snsr = conicalSensor(sat, MaxViewAngle=70, MountingLocation=[0 0 10]);
fieldOfView(snsr);
```

Add access between the conical sensor and the two ground stations.

```
acNO = access(snsr, gsNO)
```

```
acNO = 
  Access with properties:
    Sequence:  [4 1]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

```
acAmazon = access(snsr, gsAmazon)
```

```
acAmazon = 
  Access with properties:
    Sequence:  [4 2]
    LineWidth:  3
    LineColor:  [0.3922 0.8314 0.0745]
```

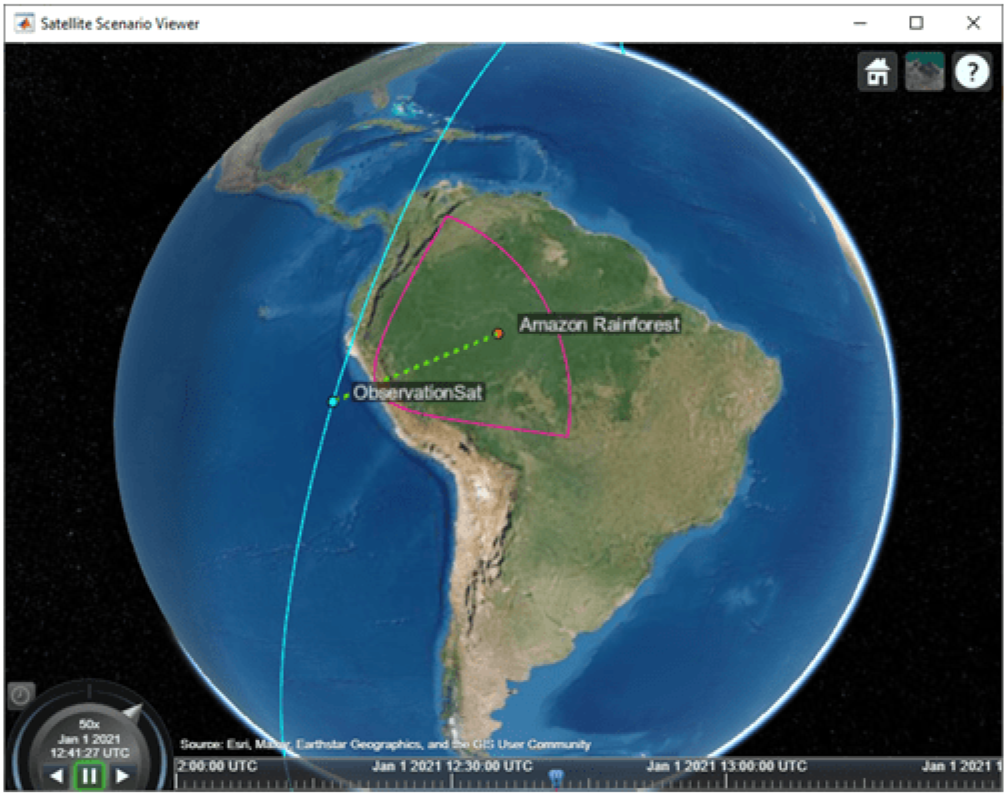
Use the `pointAt` method to associate the logged attitude `timetable` with the satellite. Parameter `ExtrapolationMethod` controls the pointing behavior outside of the `timetable` range.

```
mission.Satellite.AttitudeProfile = retime(mission.SimOutput.yout{3}.Values, ...
    seconds(uniquetol(mission.SimOutput.tout, .0001)));
pointAt(sat, mission.Satellite.AttitudeProfile, ...
    "CoordinateFrame", "inertial", "Format", "quaternion", "ExtrapolationMethod", "nadir");
```

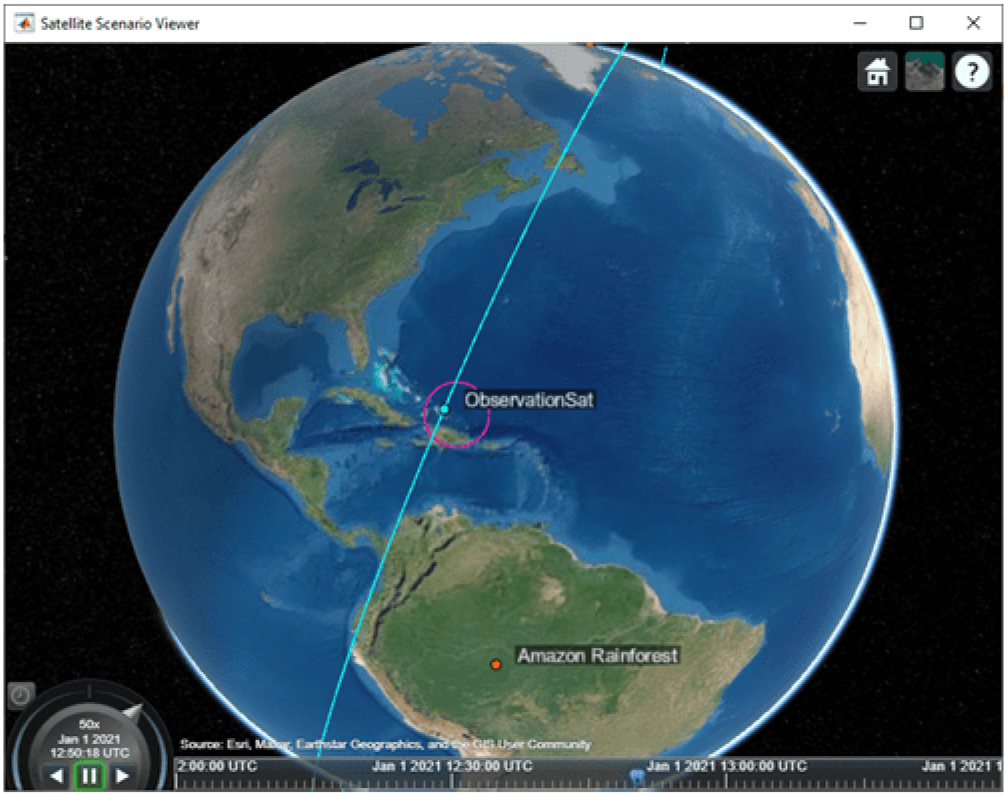
Open the **Satellite Scenario Viewer** to view and interact with the scenario.

```
viewer1 = satelliteScenarioViewer(scenario);
```

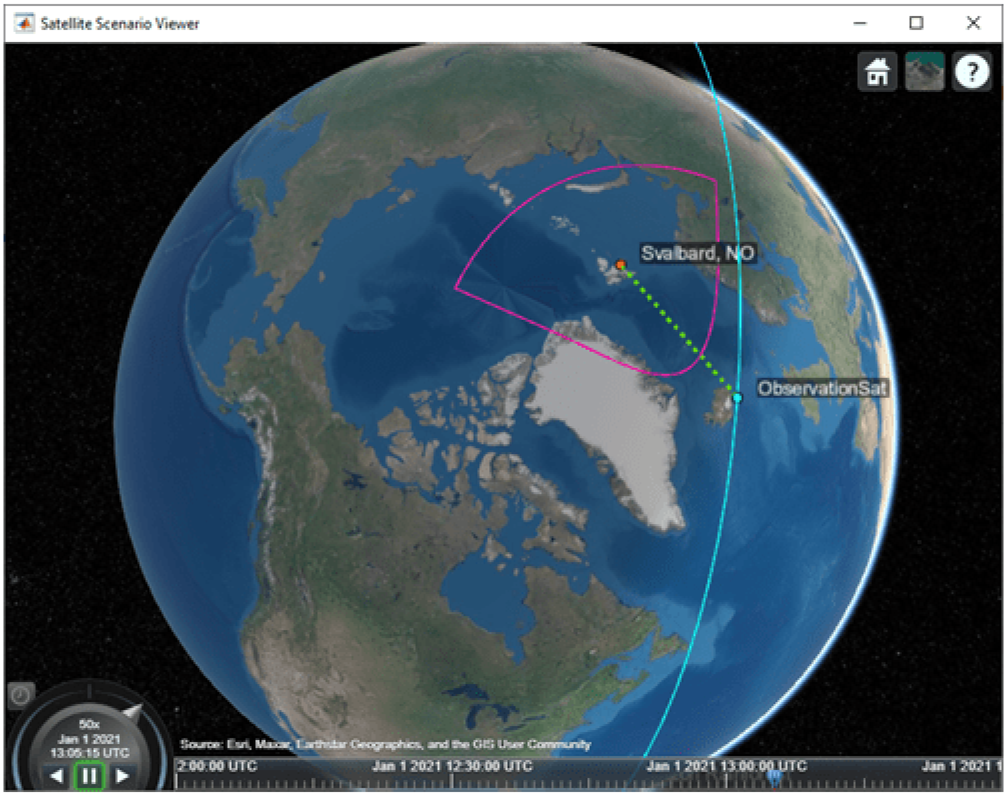
The satellite points at nadir to begin the scenario. As it nears Target 1 in the Amazon Rainforest, it slews to point and track this target.



After the imaging segment is complete, the satellite returns to pointing at nadir.



As the satellite comes into range of the arctic ground station, it slews to point at this target.



Custom Gimbal Steering

This example shows how to import custom attitude data for a simple Earth Observation satellite mission in MATLAB and Simulink, where the onboard camera is fixed to the satellite body. Another common approach is to fix the sensor on a gimbal and orient the sensor by maneuvering the gimbal, rather than the spacecraft body itself. Modify the above scenario to mount the sensor on a gimbal and steer the gimbal to perform uniform sweeps of the area directly below the satellite.

Reset the satellite to always point at nadir, overwriting the previously provided custom attitude profile.

```
delete(viewer1);
pointAt(sat, "nadir");
```

Delete the existing sensor object to remove it from the satellite and attach a new sensor with the same properties to a gimbal.

```
delete(snsr);
gim = gimbal(sat);
snsr = conicalSensor(gim, MaxViewAngle=70, MountingLocation=[0 0 10]);
fieldOfView(snsr);
```

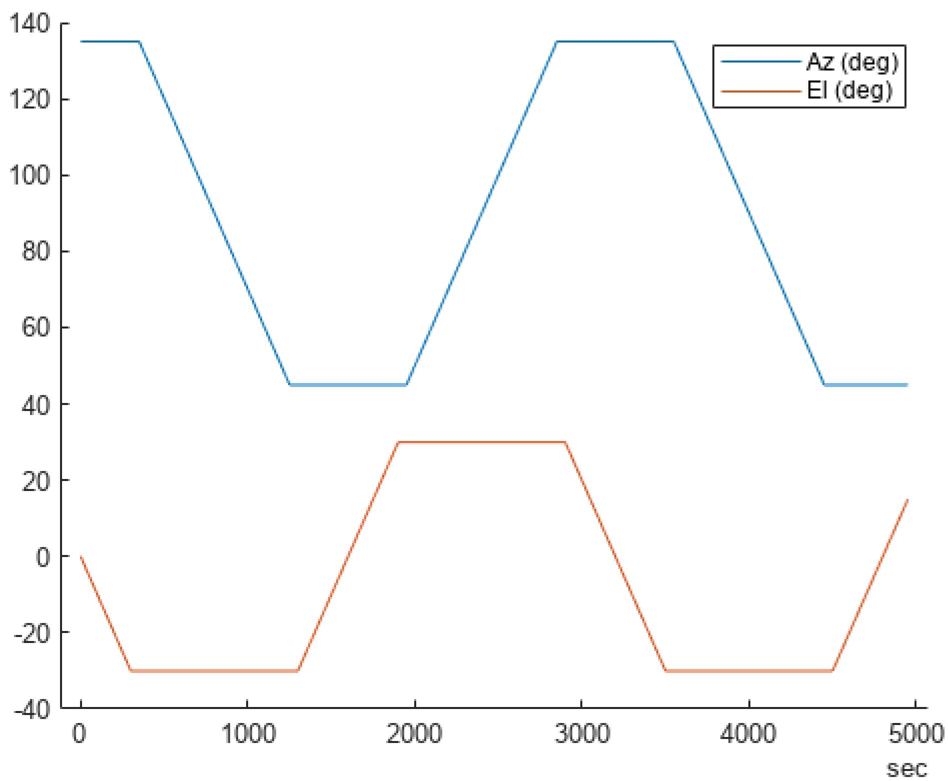
Define azimuth and elevation angles for gimbal steering to model a sweeping pattern over time below the satellite.

```
gimbalSweep.Time = seconds(1:50:5000)';

gimbalSweep.Az = [...
    45*ones(1,7),...
```

```
    45:-5:-45,...
    -45*ones(1,13),...
    -45:5:45,...
    45*ones(1,13),...
    45:-5:-45,...
    -45*ones(1,13)];
gimbalSweep.Az(end-2:end) = [];
gimbalSweep.Az = gimbalSweep.Az + 90;

gimbalSweep.El = [...
    0:-5:-30,...
    -30*ones(1,19),...
    -30:5:30,...
    30*ones(1,19),...
    30:-5:-30,...
    -30*ones(1,19),...
    -30:5:30];
gimbalSweep.El(end-2:end) = [];
```

Plot the commanded azimuth and elevation values over time.

```
figure(1)
hold on;
plot(gimbalSweep.Time', gimbalSweep.Az);
plot(gimbalSweep.Time', gimbalSweep.El);
hold off;
legend(["Az (deg)", "El (deg)"]);
```



Store the azimuth and elevation angles in a `timetable`.

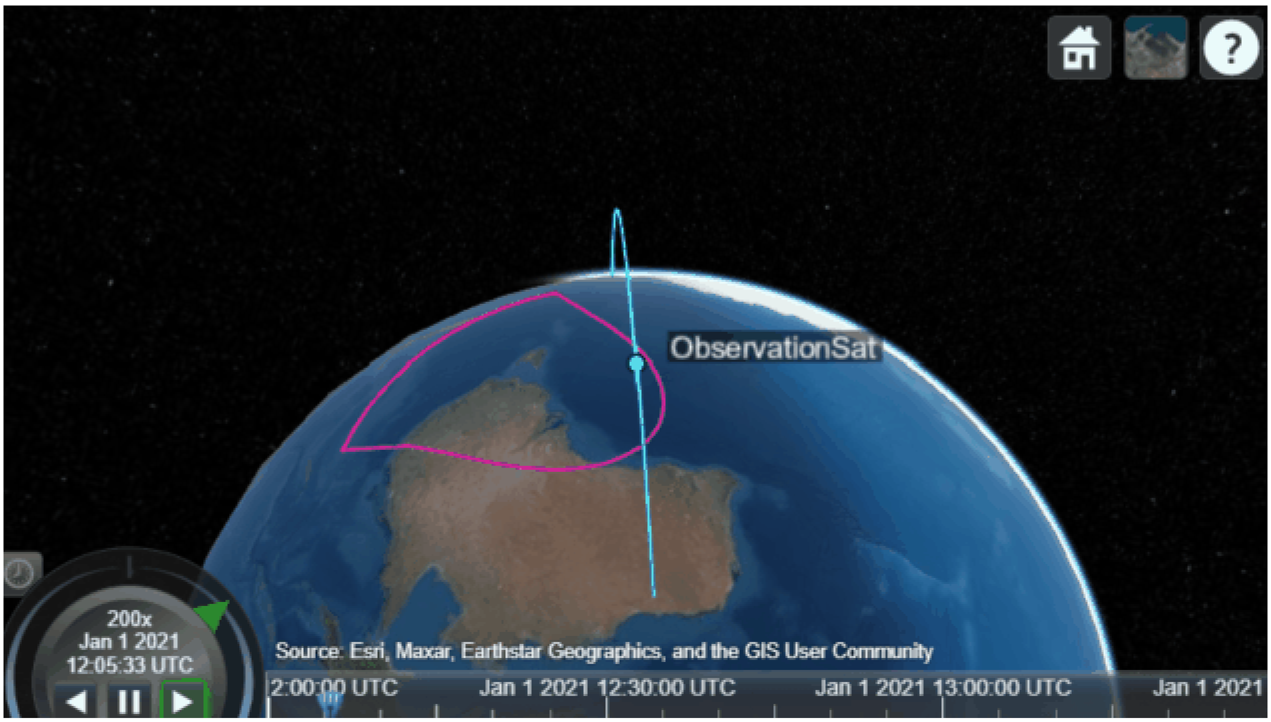
```
gimbalSweep.TT = timetable(gimbalSweep.Time, [gimbalSweep.Az', gimbalSweep.El']);
```

Steer the gimbal with the `timetable`. The gimbal returns to its default orientation for timesteps that are outside of the provided data.

```
pointAt(gim, gimbalSweep.TT);
```

View the updated scenario in the **Satellite Scenario Viewer**.

```
viewer2 = satelliteScenarioViewer(scenario);
```



See Also

Blocks
Spacecraft Dynamics | Attitude Profile

Objects
`satelliteScenario`

Model-Based Systems Engineering for Space-Based Applications

This example provides an overview of the **CubeSat Model-Based System Engineering Project** template, available from the Simulink® start page, under Aerospace Blockset™. It demonstrates how to model a space mission architecture in Simulink with System Composer™ and Aerospace Blockset for a 1U CubeSat in low Earth orbit (LEO). The CubeSat's mission is to image MathWorks Headquarters in Natick, Massachusetts, at least once per day. The project references the Aerospace Blockset *CubeSat Simulation Project*, reusing the vehicle dynamics, environment models, data dictionaries, and flight control system models defined in that project.

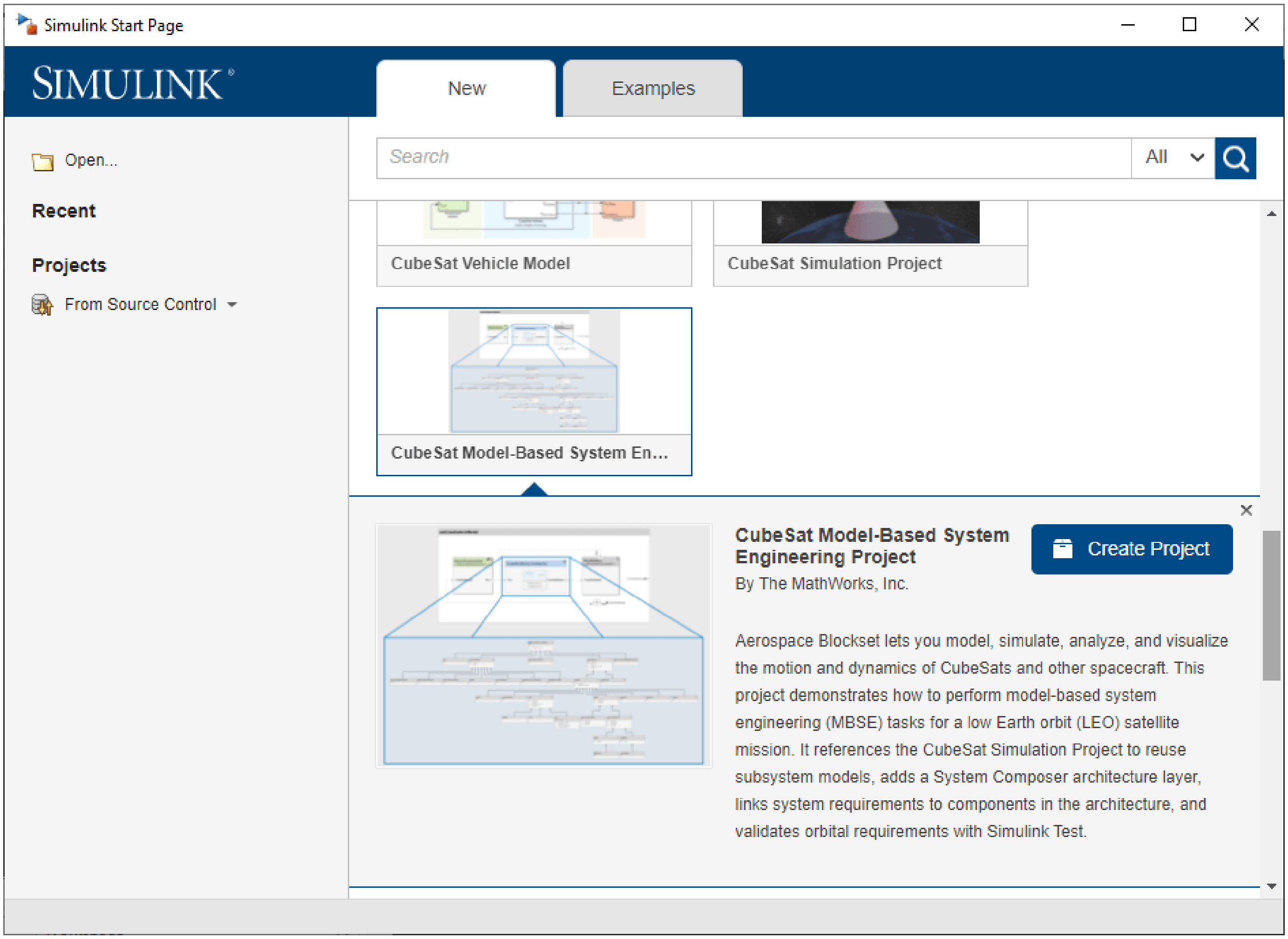
This project demonstrates how to:

- Define system level requirements for a CubeSat mission in Simulink
- Compose a system architecture for the mission in System Composer
- Link system-level requirements to components in the architecture with Requirements Toolbox™
- Model vehicle dynamics and flight control systems with Aerospace Blockset
- Validate orbital requirements using mission analysis tools and Simulink Test™

Open the Project

To create a new instance of the **CubeSat Model-Based System Engineering Project**, select **Create Project** in the Simulink start page. When the project is loaded, an architecture model for the CubeSat opens.

```
open("asbCubeSatMBSEProject.sltx");
```



Define System-Level Requirements

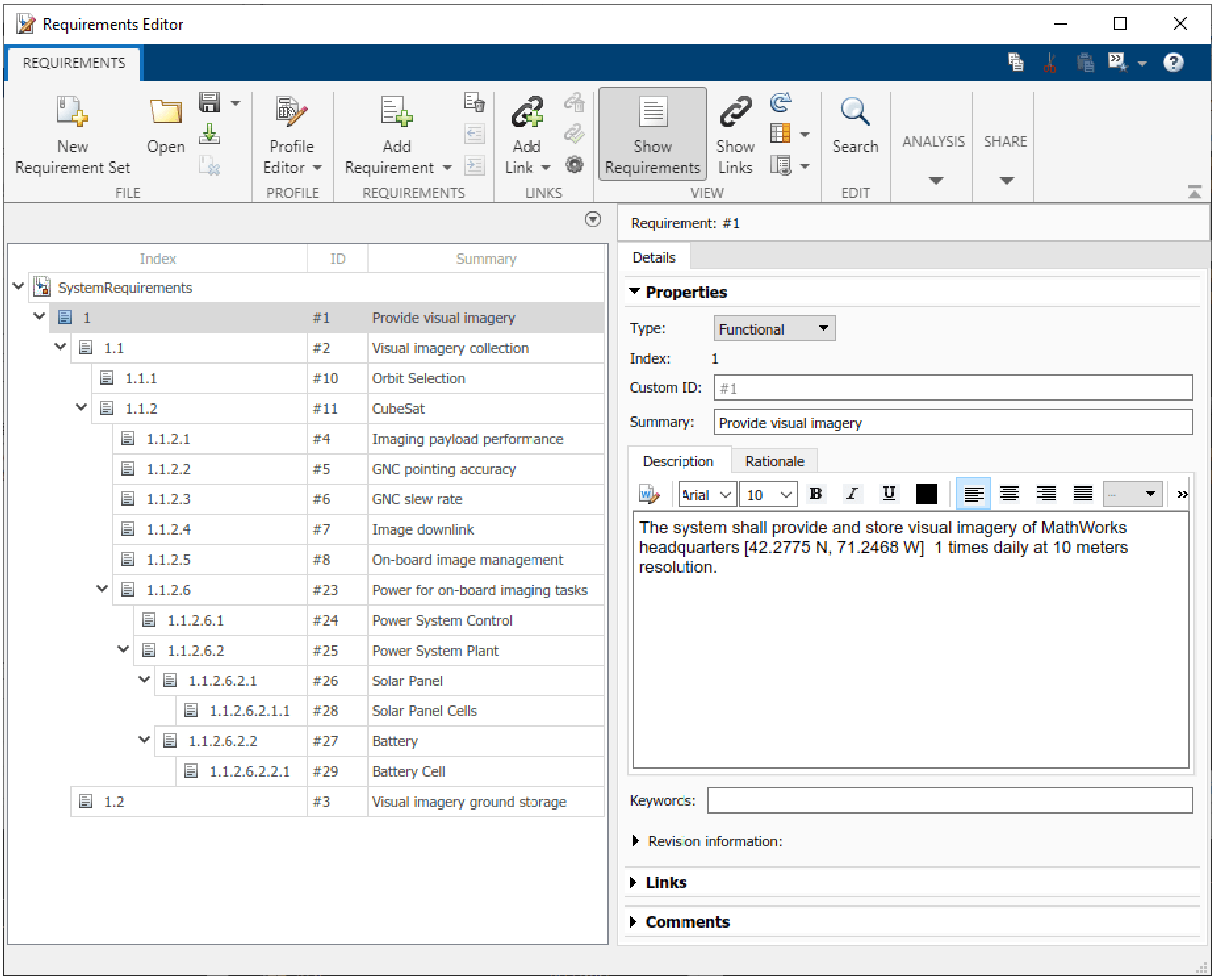
Define a set of system-level requirements for the mission. You can import these requirements from third-party requirement management tools such as ReqIF (Requirements Interchange Format) files or author them directly in the Requirements Editor.

This example contains a set of system-level requirements stored in *SystemRequirements.slreqx*. Open this requirement specification file in the **Requirements Editor**. Access the **Requirements Editor** from the **Apps** tab or by double-clicking on *SystemRequirements.slreqx* in the project folder browser.

Our top level requirement for this mission is:

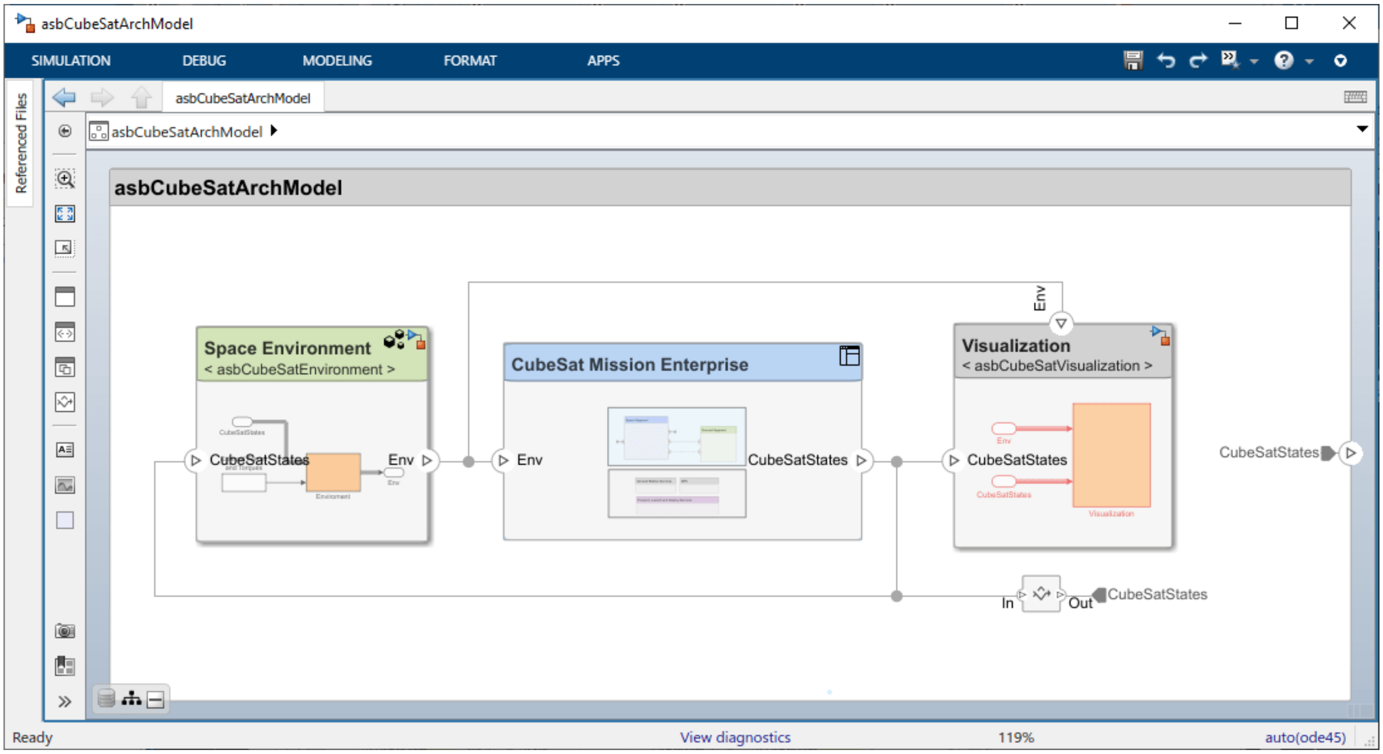
1 The system shall provide and store visual imagery of MathWorks® headquarters [42.2775 N, 71.2468 W] once daily at 10 meters resolution.

Additional requirements are decomposed from this top-level requirement to create a hierarchy of requirements for the architecture.



Compose a System Architecture

System Composer enables the specification and analysis of architectures for model-based systems engineering. Use the system-level requirements defined above to guide the creation of an architecture model in System Composer. The architecture in this example is based on *CubeSat Reference Model (CRM)* developed by the International Council on Systems Engineering (INCOSE) Space Systems Working Group (SSWG) [1].



The architecture is composed of components, ports, and connectors. A component is a part of a system that fulfills a clear function in the context of the architecture. It defines an architectural element, such as a system, subsystem, hardware, software, or other conceptual entity.

Ports are nodes on a component or architecture that represent a point of interaction with its environment. A port permits the flow of information to and from other components or systems. Connectors are lines that provide connections between ports. Connectors describe how information flows between components in an architecture.

Extend the Architecture with Stereotypes and Interfaces

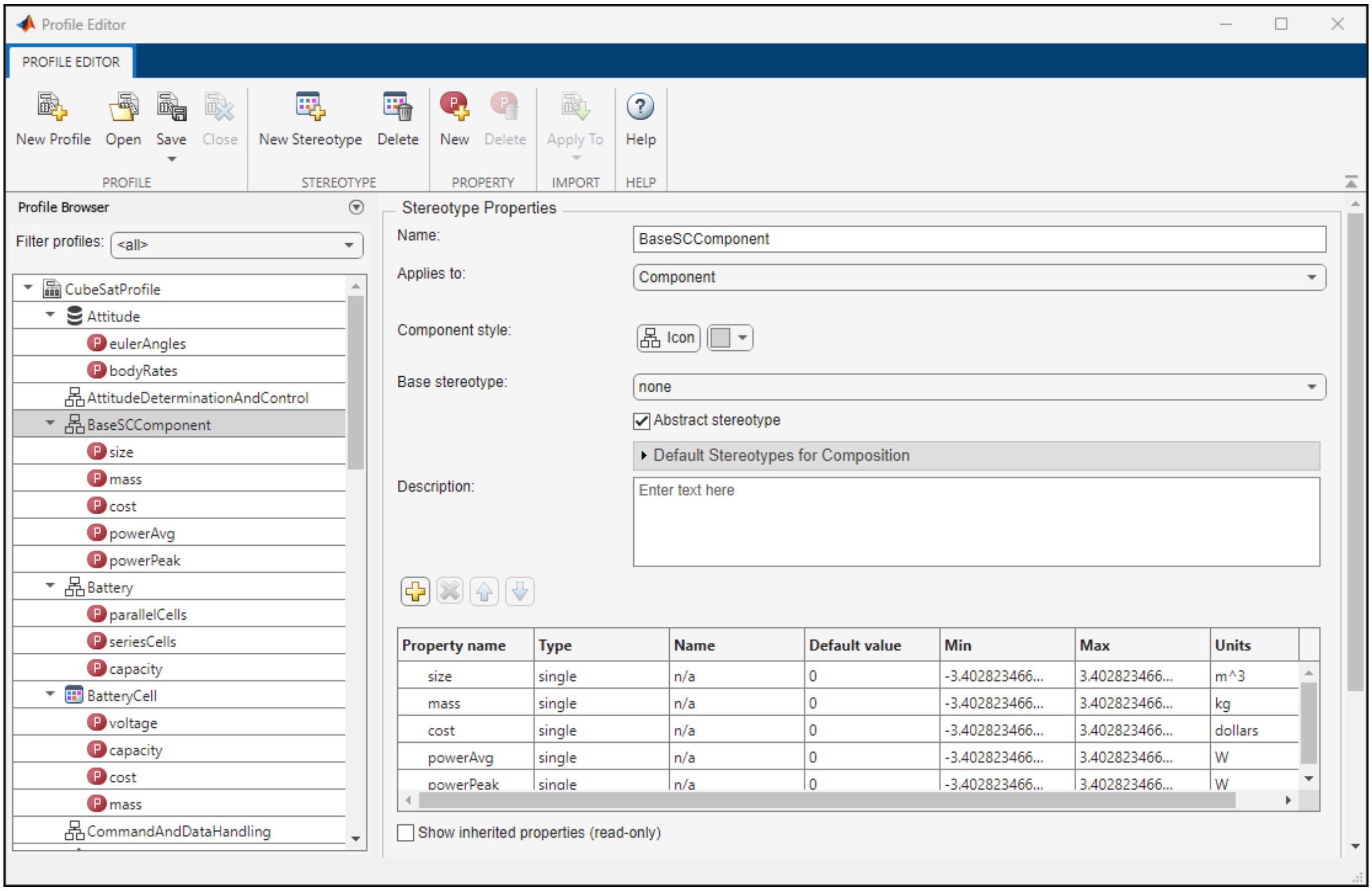
You can add additional levels of detail to an architecture using stereotypes and interfaces.

Stereotypes

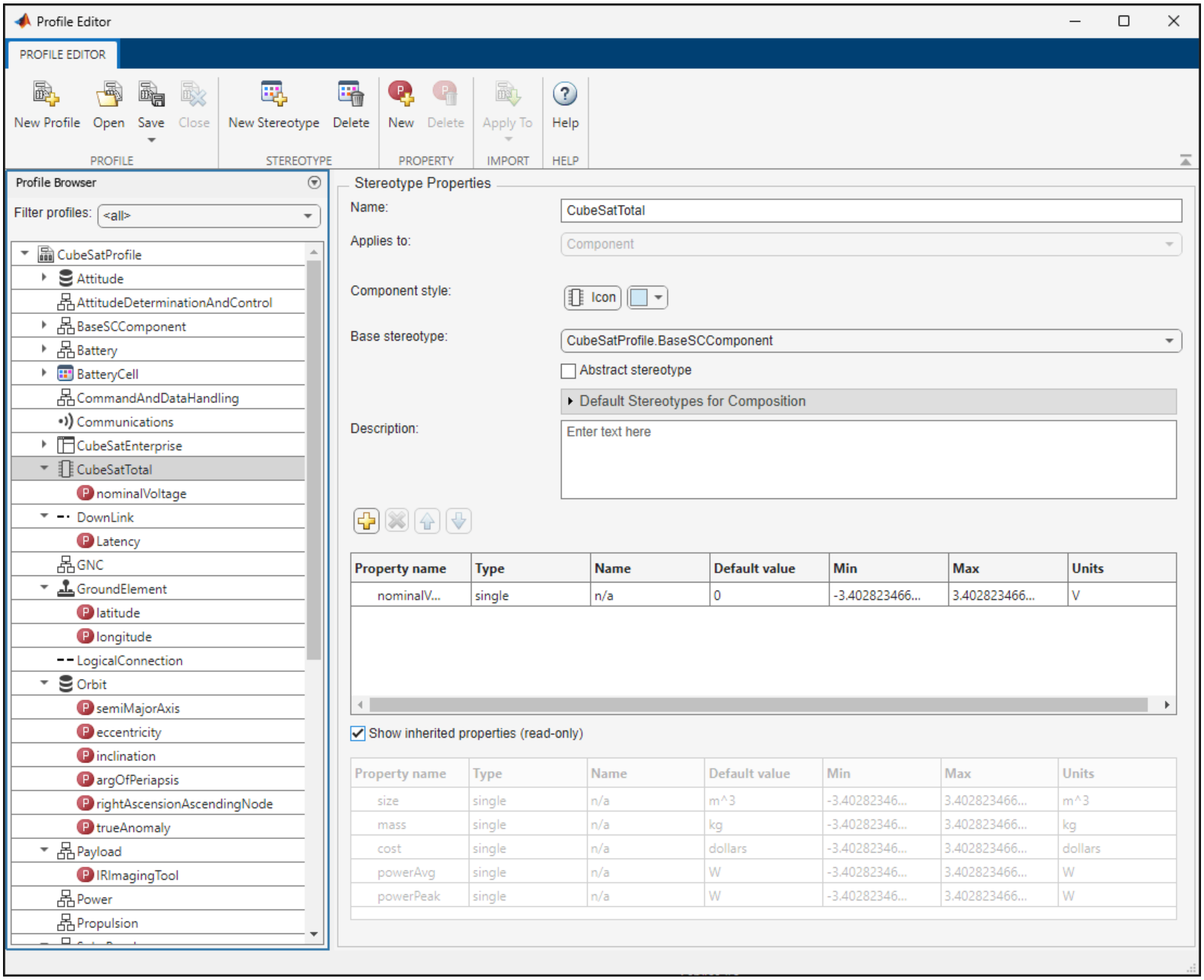
Stereotypes extend the architectural elements by adding domain-specific metadata to each element. Stereotypes are applied to components, connectors, ports, and other architectural elements to provide these elements with a common set of properties such as mass, cost, power, etc.

Packages of stereotypes used by one or more architectures are stored in profiles. This example includes a profile of stereotypes called *CubeSatProfile.xml*. To view, edit, or add new stereotypes to the profile, open this profile in the **Profile Editor** from the **Modeling** Tab.

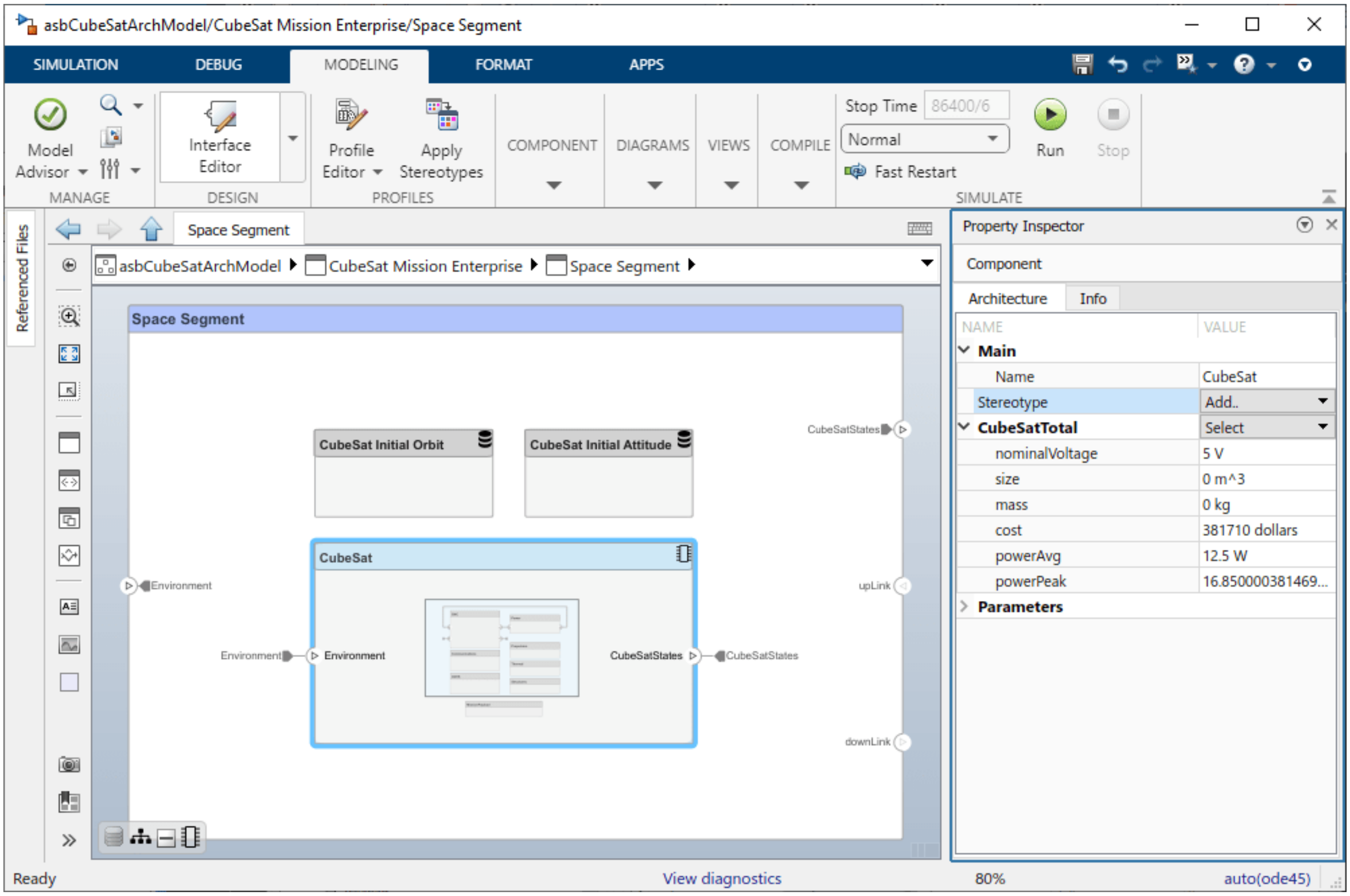
This profile defines a set of stereotypes that are applied to components and connectors in the CubeSat architecture.



Stereotypes can also inherit properties from abstract base stereotypes. For example, `BaseSCComponent` in the profile above contains properties for size, mass, cost, and power demand. We can add another stereotype to the profile, `CubeSatTotal`, and define `BaseSCComponent` as its base stereotype. `CubeSatTotal` adds in its own property, nominalVoltage, but also inherits properties from its base stereotype.

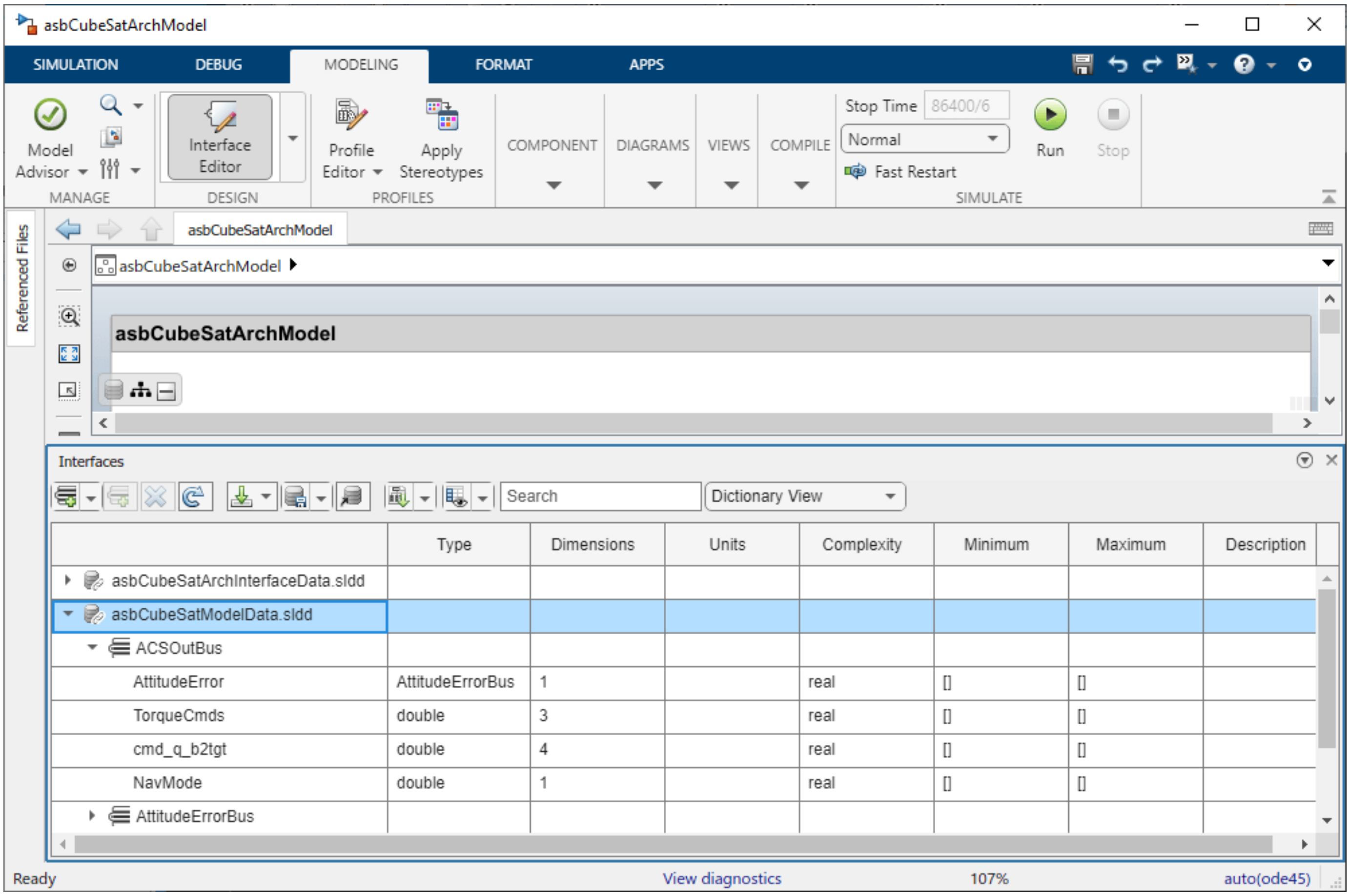


In the architecture model, apply the `CubeSatTotal` stereotype to CubeSat system component (`asbCubeSatArchModel/CubeSat Mission Enterprise/Space Segment/CubeSat`). Select the component in the model. In the Property Inspector, select the desired stereotype from the drop-down window. Next, set property values for the CubeSat component.



Interfaces

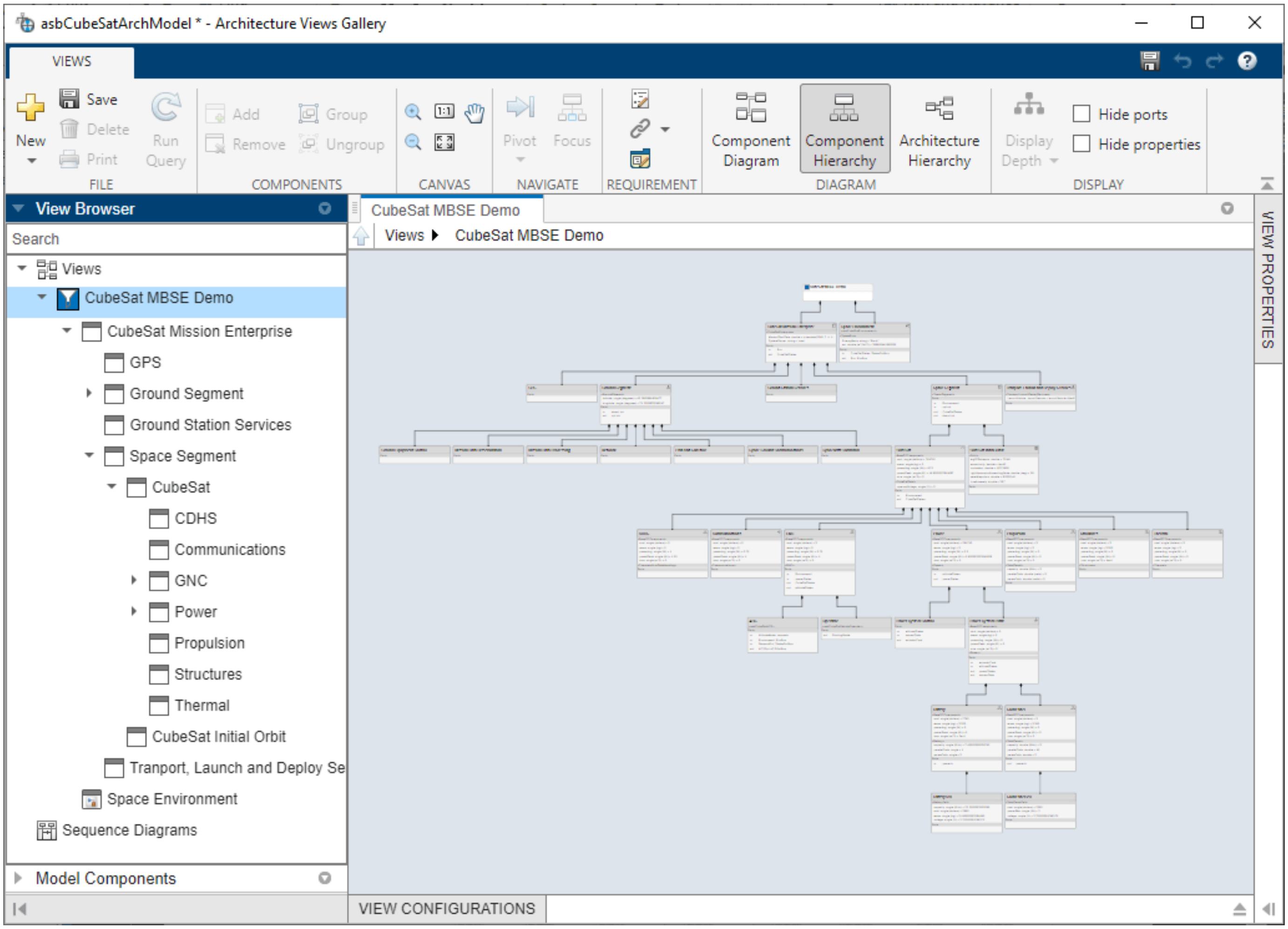
Data interfaces define the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal. Create and manage interfaces from the **Interface Editor**. Existing users of Simulink can draw a parallel between interfaces in System Composer and buses in Simulink. In fact, buses can be used to define interfaces (and vice versa). For example, the data dictionary *asbCubeSatModelData.sldd* contains several bus definitions, including `ACSOutBus`, that can be viewed in the **Interface Editor** and applied to architecture ports.



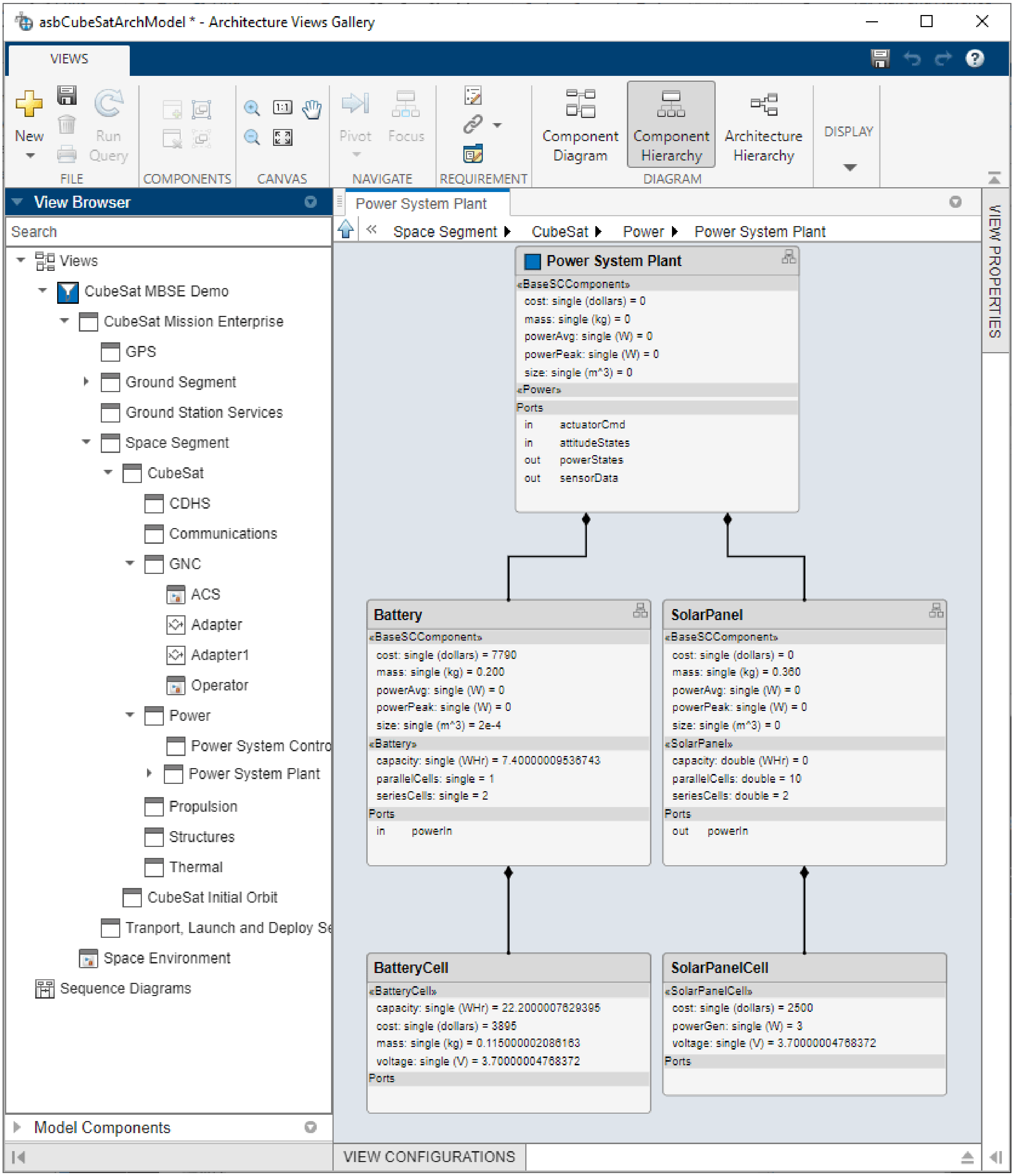
Visualize the System with Architecture Views

Now that we have implemented our architecture using components, stereotypes, ports, and interfaces, we can visualize our system with an architecture view. In the **Modeling** Tab, select **Views**.

Use the **Component Hierarchy** view to show our system component hierarchy. Each component also lists its stereotype property values and ports.

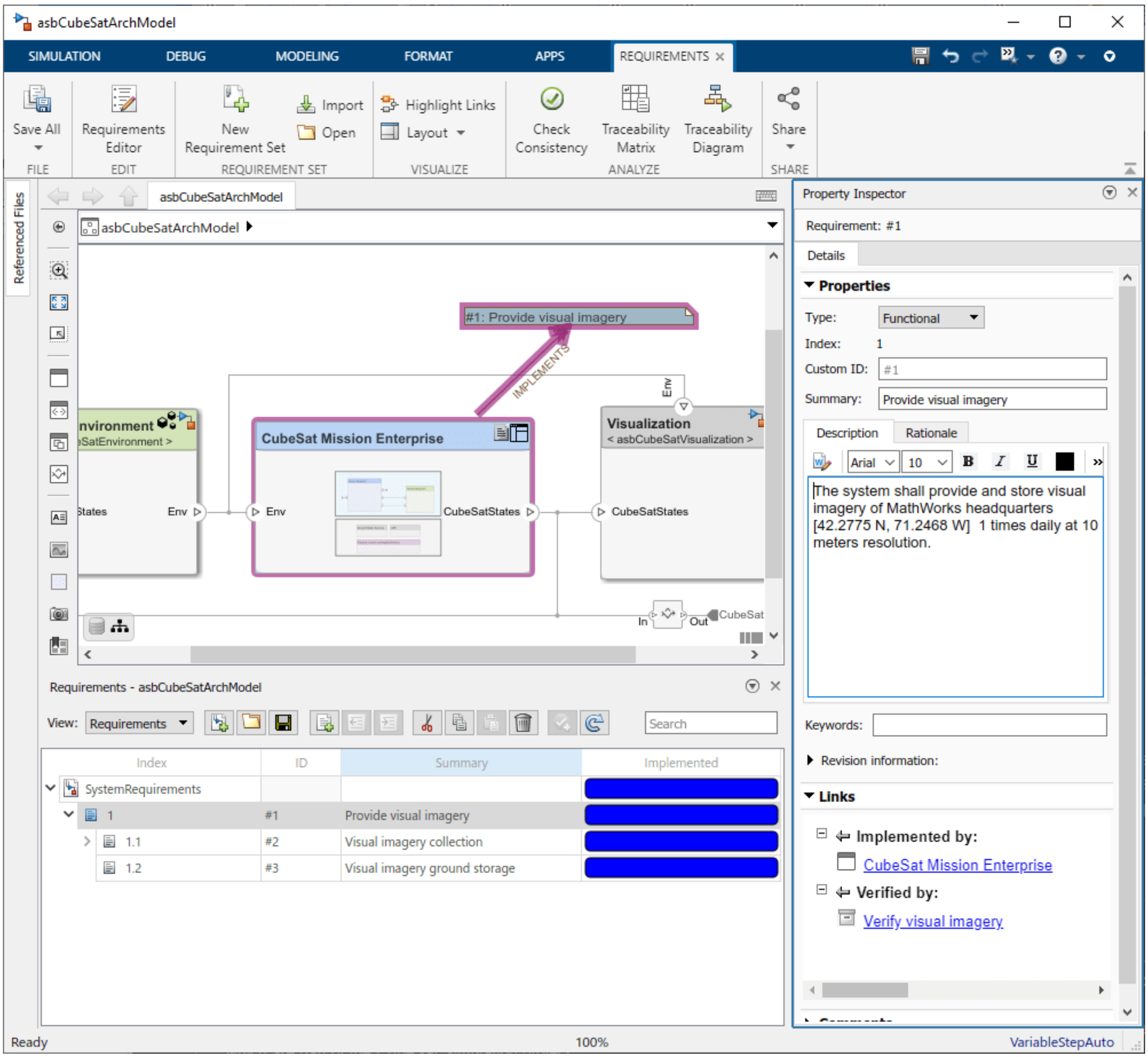


You can also view the hierarchy at different depths of the architecture. For example, navigate to the `Power System Plant` component of the architecture by double-clicking the component in the **View Browser**.


asbCubeSatArchModel * - Architecture Views Gallery
VIEWS
New
Run Query
FILE
COMPONENTS
CANVAS
Pivot
Focus
NAVIGATE
REQUIREMENT
Component Diagram
Component Hierarchy
Architecture Hierarchy
DIAGRAM
DISPLAY
View Browser
Search
Views
CubeSat MBSE Demo
CubeSat Mission Enterprise
GPS
Ground Segment
Ground Station Services
Space Segment
CubeSat
CDHS
Communications
GNC
ACS
Adapter
Adapter1
Operator
Power
Power System Contro
Power System Plant
Propulsion
Structures
Thermal
CubeSat Initial Orbit
Tranport, Launch and Deploy Se
Space Environment
Sequence Diagrams
Model Components
Power System Plant
Space Segment
CubeSat
Power
Power System Plant
Power System Plant
«BaseSCComponent»
cost: single (dollars) = 0
mass: single (kg) = 0
powerAvg: single (W) = 0
powerPeak: single (W) = 0
size: single (m^3) = 0
«Power»
Ports
in actuatorCmd
in attitudeStates
out powerStates
out sensorData
Battery
«BaseSCComponent»
cost: single (dollars) = 7790
mass: single (kg) = 0.200
powerAvg: single (W) = 0
powerPeak: single (W) = 0
size: single (m^3) = 2e-4
«Battery»
capacity: single (WHr) = 7.40000009536743
parallelCells: single = 1
seriesCells: single = 2
Ports
in powerIn
SolarPanel
«BaseSCComponent»
cost: single (dollars) = 0
mass: single (kg) = 0.360
powerAvg: single (W) = 0
powerPeak: single (W) = 0
size: single (m^3) = 0
«SolarPanel»
capacity: double (WHr) = 0
parallelCells: double = 10
seriesCells: double = 2
Ports
out powerIn
BatteryCell
«BatteryCell»
capacity: single (WHr) = 22.2000007629395
cost: single (dollars) = 3895
mass: single (kg) = 0.115000002086163
voltage: single (V) = 3.70000004768372
Ports
SolarPanelCell
«SolarPanelCell»
cost: single (dollars) = 2500
powerGen: single (W) = 3
voltage: single (V) = 3.70000004768372
Ports
VIEW CONFIGURATIONS
VIEW PROPERTIES

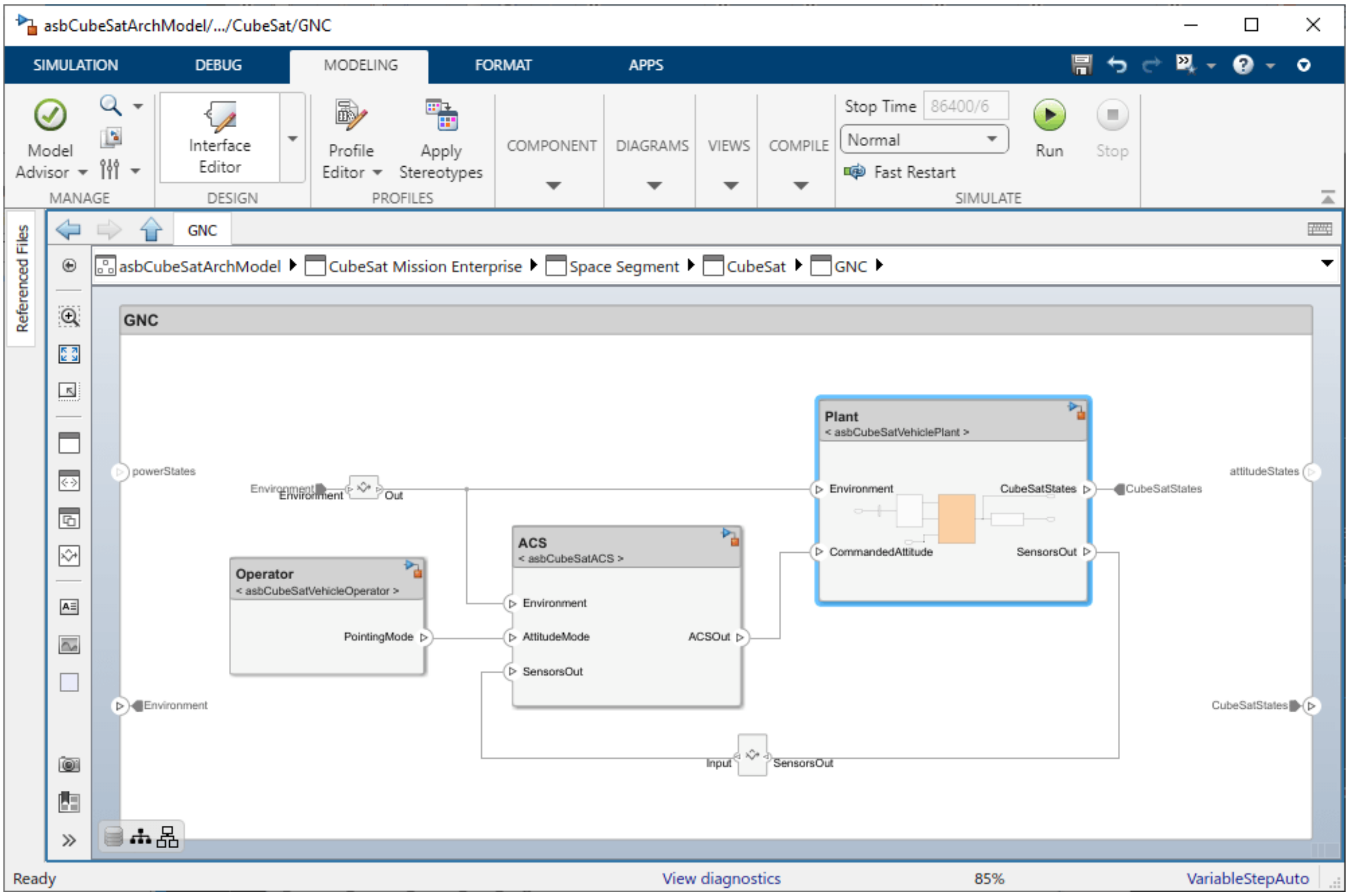
Link Requirements to Architecture Components

To link requirements to the architectural elements that implement them, use the **Requirements Manager.** Drag the requirement onto the corresponding component, port, or interface. Using this linking mechanism, we can identify how requirements are met in the architecture model. The column labeled "Implemented" in the **Requirements Manager** shows whether a textual requirement has been linked to a component in the given model. For example, our top-level requirement "Provide visual imagery" is linked to our top-level component `CubeSat Mission Enterprise` with decomposed requirements linked to respective decomposed architectural components.

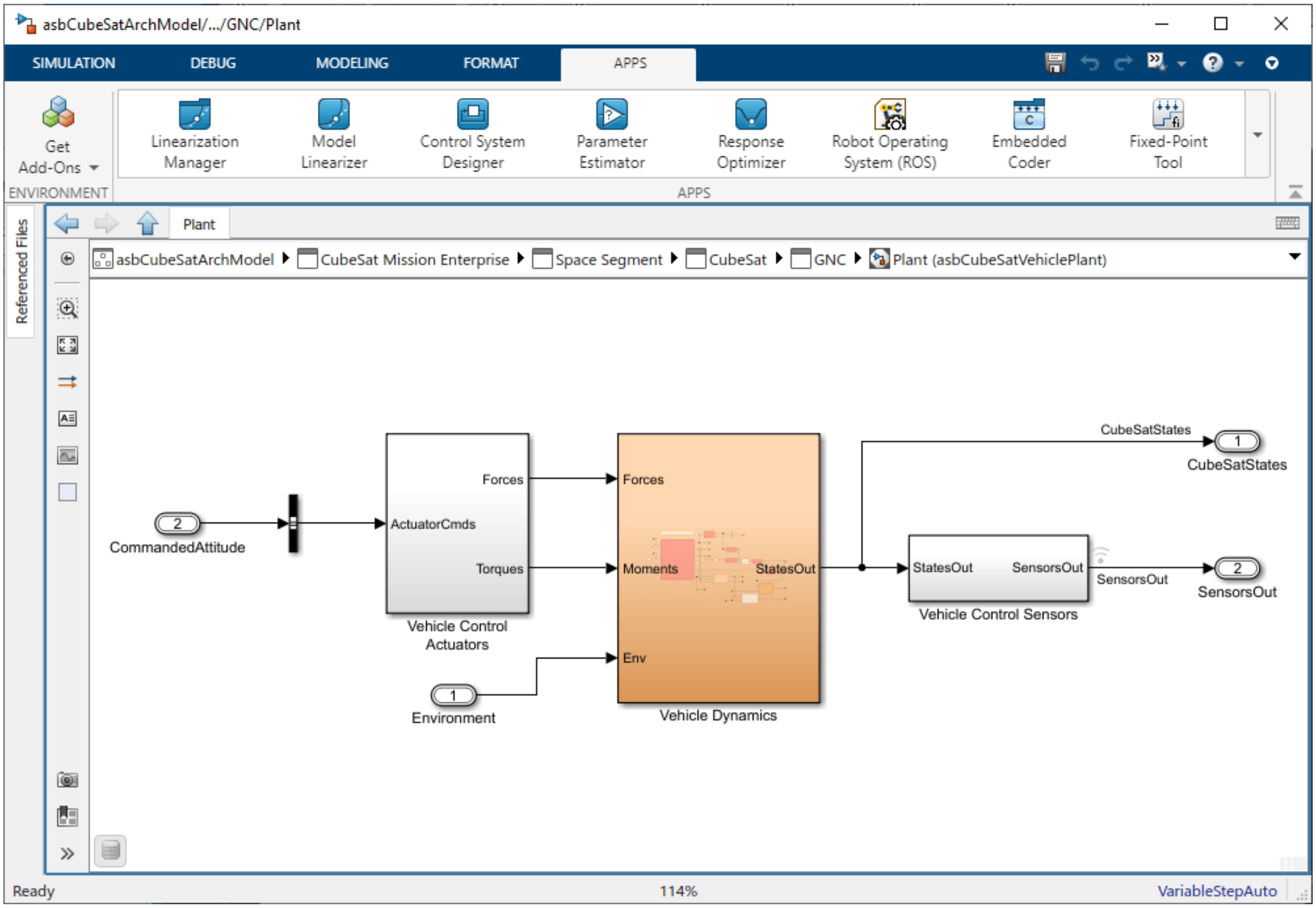


Connecting the Architecture to Design Models

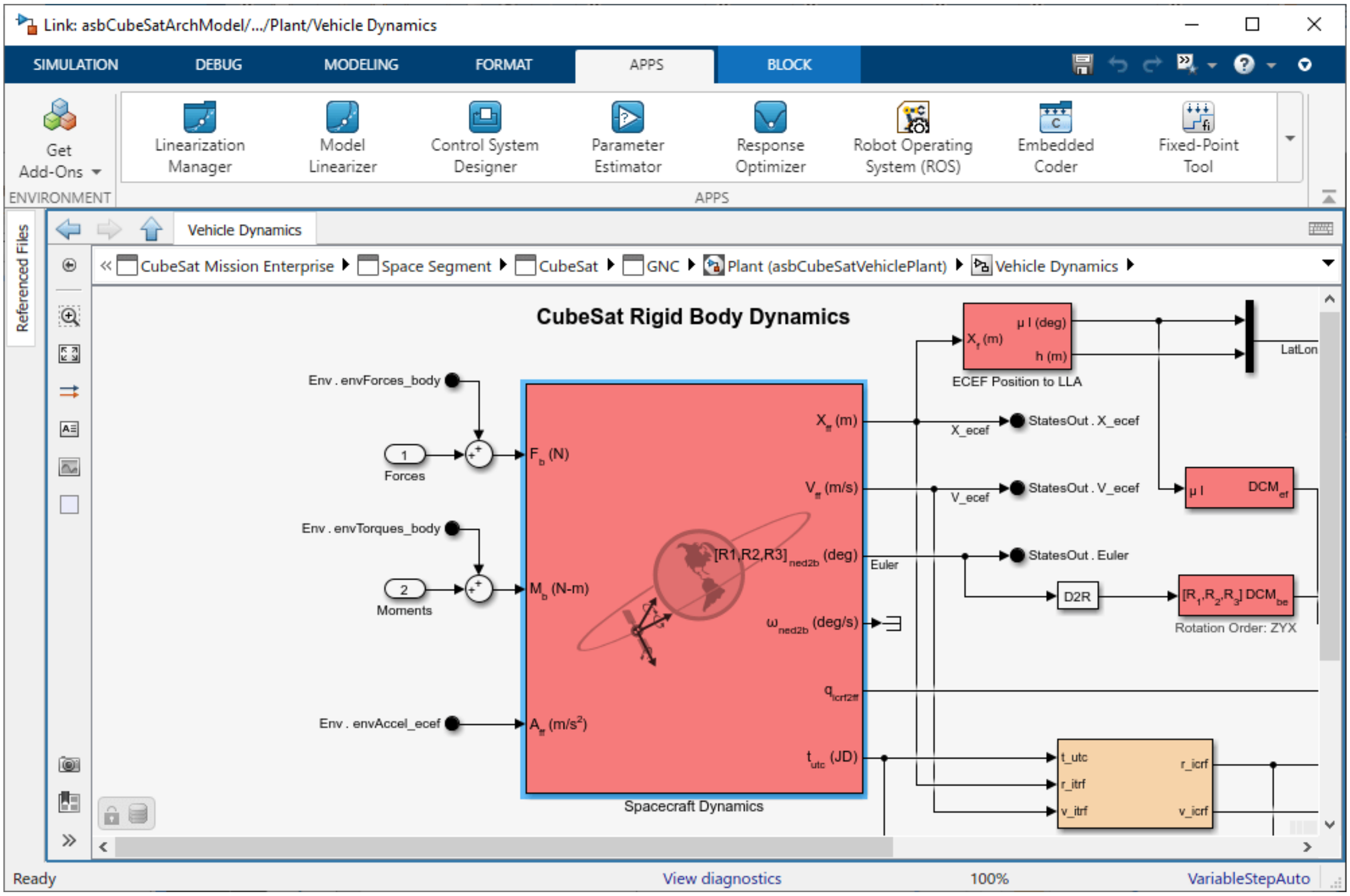
As the design process matures through analysis and other systems engineering processes, we can begin to populate our architecture with dynamics and behavior models. System Composer is built as a layer on top of Simulink, which enables Simulink models to be directly referenced from the components we have created. We can then simulate our architecture model as a Simulink model and generate results for analysis. For example, the `GNC subsystem` component contains 3 Simulink model references that are part of the *CubeSat Simulation Project*.



Double-click these reference components to open the underlying Simulink models. Notice that the interfaces defined in the architecture map to bus signals in the Simulink model.

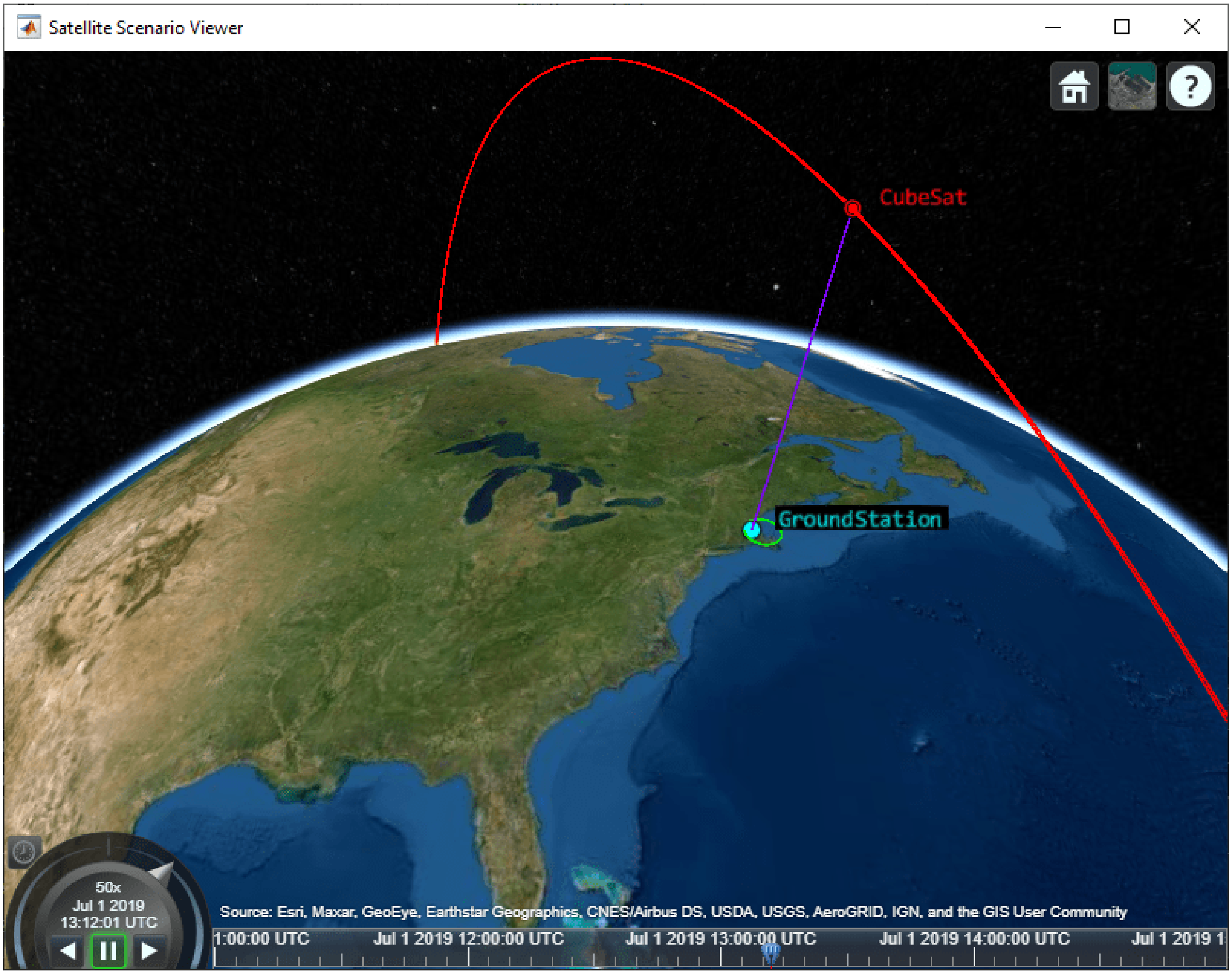


This example uses the **Spacecraft Dynamics** block from Aerospace Blockset to propagate the CubeSat orbit and rotational states.



Simulate System Architecture to Validate Orbital Requirements

We can use simulation to verify our system-level requirements. In this scenario, our top level requirement states that the CubeSat onboard camera captures an image of MathWorks Headquarters at [42.2775 N, 71.2468 W] once daily at 10 meters resolution. We can manually validate this requirement with various mission analysis tools. For examples of these analyses, click on the project shortcuts *Analyze with Mapping Toolbox* and *Analyze with Satellite Scenario*.

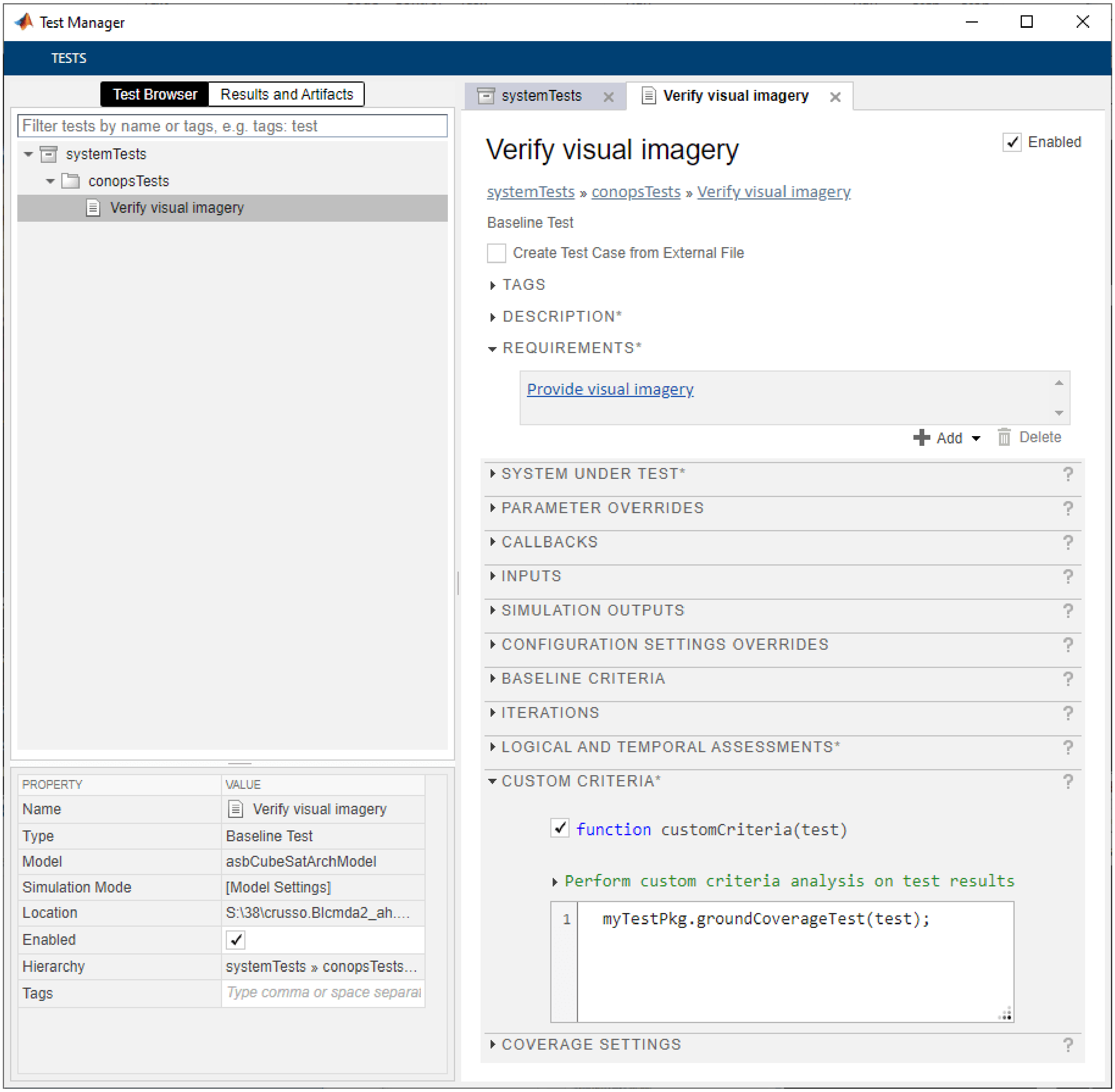


The satellite scenario created in the *Analyze with Satellite Scenario* shortcut example is shown above.

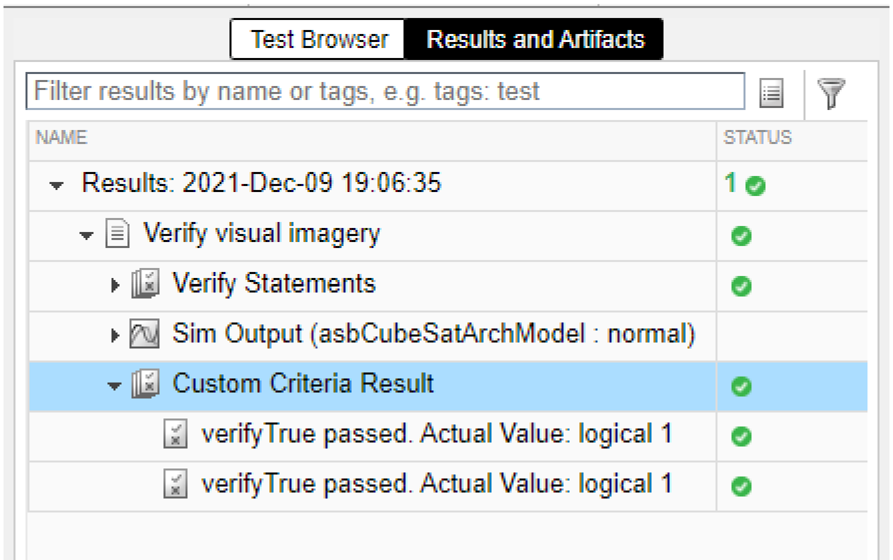
Validate Orbital Requirements Using Simulink Test

Although we can use MATLAB to visualize and analyze the CubeSat behavior, we can also use Simulink Test to build test cases. This test case automates the requirements-based testing process by using the testing framework to test whether our CubeSat orbit and attitude meet our high-level requirement. The test case approach enables us to create a scalable, maintainable, and flexible testing infrastructure based on our textual requirements.

This example contains a test file *systemTests.mldatx*. Double-click this file in the project folder browser to view it in the **Test Manager**. Our test file contains a test to verify our top-level requirement. The "Verify visual imagery" testpoint is mapped to the requirement "Provide visual imagery" and defines a MATLAB function to use as custom criteria for the test. While this test case is not a comprehensive validation of our overall mission, it is useful during early development to confirm our initial orbit selection is reasonable, allowing us to continue refining and adding detail to our architecture.



Run the test point in the **Test Manager** and confirm that the test passes. Passing results indicate that the CubeSat onboard camera as visibility to the imaging target during the simulation window.



References

[1] “Space Systems Working Group.” INCOSE, 2019, https://www.incose.org/communities/working-groups-initiatives/space-systems.

See Also

Orbit Propagator | Spacecraft Dynamics | Attitude Profile

Related Examples

- “CubeSat Simulation Project” on page 2-75
- “Compose and Analyze Systems Using Architecture Models” (System Composer)

High Precision Orbit Propagation of the International Space Station

This example shows how to propagate the orbit of the International Space Station (ISS) using high precision numerical orbit propagation with Aerospace Blockset™. It uses the Orbit Propagator block to calculate the ISS trajectory for 24 hours. The example then compares position and velocity states to publicly available ISS trajectory data available from NASA Trajectory Operations and Planning (TOPO) flight controllers at Johnson Space Center.

The Orbit Propagator block models translational dynamics of spacecraft using numerical integration. It computes the position and velocity of one or more spacecraft over time. For the most accurate results, use a variable step solver with low tolerance settings (less than 1e-9). Depending on your mission requirements, you can speed up simulation run time by using larger tolerances. Doing so might impact the accuracy of the solution. To propagate orbital states, the block uses the gravity model selected for the current central body. The block also includes atmospheric drag (for Earth orbits), gravitational effects of celestial bodies other than the central body, and solar radiation pressure. The block also takes into account external perturbing accelerations that you provide as inputs to the block.

In this example, use ISS International Celestial Reference Frame (ICRF) position and velocity states directly from the referenced NASA public distribution file [1] to initialize an ISS orbit in the Orbit Propagator block. Use the EGM2008 spherical harmonic gravity model to model Earth gravity and the NRLMSISE-00 atmospheric model to compute atmospheric drag, including the effects of space weather (solar flux and geomagnetic indices) on atmospheric density. Calculate third body gravity point-mass contributions for the Moon, Sun, Jupiter. Lastly, include effects of solar radiation pressure (SRP) assuming a spherical satellite geometry. Perturbations due to third body gravity and solar radiation pressure are near-negligable in a low Earth orbit like that of the ISS. However, to demostrate possible configurations for high precision orbit propagation, they are included in this example. Perturbations required for accurate trajectory modeling vary depending on the application and orbital regime. Determine which perturbations to include on a case-by-case basis for your mission.

Run the model for 24 hours, and compare against the published NASA trajectory data.

Set Mission Initial Conditions

This example uses MATLAB® structures to organize mission data. These structures make accessing data later in the example more intuitive. They also help declutter the global base workspace.

The referenced ISS ephemeris data file [1] provided by NASA begins on January 3, 2022 at 12:00:00.000 UTC.

```
mission.StartDate = datetime(2022, 1, 3, 12, 0, 0);
mission.Duration  = hours(24);
```

The file contains ICRF position (km) and velocity (km/s) data sampled every 4 minutes.

```
iss.X0_icrf = [-1325.896391725290 5492.890955896010 3762.423747679220]; % km
iss.V0_icrf = [-4.87470128630892  -4.10251688094599 4.26428812476909];  % km/s
```

It also contains mass (kg), drag area (m^2), and drag coefficient data corresponding with the epoch.

```
iss.Mass      = 459023.0; % kg
iss.DragArea  = 1951.0;   % m^2
iss.DragCoeff = 2.0;
```

The file also includes solar radiation area (m^2) and a solar radiation coefficient; however these values are zero because SRP is not prominent in low Earth orbit (LEO) where the ISS operates. Despite their trivial impact on the resultant trajectory, we will include solar radiation pressure calculations in this example to fully demonstrate high precision orbit propagation. SRP is more prominent at higher orbital regimes.

```
iss.SRPArea  = 1500;         % m^2
iss.SRPCoeff = 1.8;
iss.P_sr     = 4.5344321e-6; % N/m^2
```

High Precision Orbit Propagation Algorithm

The Orbit Propagator block performs high precision numerical orbit propagation using Cowell's method. To compute the position and velocity at each time step of the simulationn the inertial (ICRF) frame, Earth gravitational acceleration is summed with all perturbating accelerations and double-integrated.

$$\vec{a}_{\mathrm{icrf}} = \vec{a}_{\mathrm{gravity}} + \vec{a}_{\mathrm{drag}} + \vec{a}_{\mathrm{3rd\,body}} + \vec{a}_{\mathrm{srp}}$$

$$\vec{a}_{\mathrm{icrf}} \xRightarrow[\mathrm{integrate}]{} \vec{v}_{\mathrm{icrf}}, \vec{r}_{\mathrm{icrf}}$$

where:

$\vec{a}_{\mathrm{gravity}}$ is the acceleration due to Earth gravity.

$\vec{a}_{\mathrm{drag}}$ is the acceleration due to atmospheric drag.

$\vec{a}_{\mathrm{3rd\,body}} = \vec{a}_{\mathrm{Moon}} + \vec{a}_{\mathrm{Sun}} + \vec{a}_{\mathrm{Mercury}} + \vec{a}_{\mathrm{Venus}} + \vec{a}_{\mathrm{Mars}} + \vec{a}_{\mathrm{Jupiter}} + \vec{a}_{\mathrm{Saturn}} + \vec{a}_{\mathrm{Uranus}} + \vec{a}_{\mathrm{Neptune}}$ is the acceleration due to gravity of the Moon, Sun, and planets of the solar system. The correct list of third bodies to include depends on the central body, orbit, and application.

$\vec{a}_{\mathrm{srp}}$ is the acceleration due to solar radiation pressure.

Earth Gravity

The gravity model for Earth in this example is the EGM2008 spherical harmonic gravity model. This model accounts for zonal, sectoral, and tesseral harmonics. For reference, the second-degree zeroth order zonal harmonic J_2, which accounts for the oblateness of Earth, is $-C_{2,0}$. Spherical Harmonic models accounts for harmonics up to max degree $l = l_{\mathrm{max}}$. For EGM2008, $l_{\mathrm{max}} = 2159$.

Spherical harmonic gravity is computed in the fixed frame (ff) coordinated system (International Terrestrial Reference Frame (ITRF), in the case of Earth). Numerical integration, however, is always performed in the inertial ICRF coordinate system. Therefore, at each timestep, position and velocity states are transformed into the fixed-frame, gravity is calculated in the fixed-frame, and the resulting acceleration is transformed into the inertial frame.

$$\vec{a}_{\mathrm{gravity}} = -\frac{\mu}{r^3}\overrightarrow{r_{\mathrm{icrf}}} + \mathrm{ff2icrf}\left(\vec{a}_{\mathrm{nonspherical}}\right)$$

where:

$$\vec{a}_{\text{nonspherical}} = \left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U - \frac{r_{\text{ff}k}}{r^2\sqrt{r_{\text{ff}i}^2 + r_{\text{ff}j}^2}}\frac{\partial}{\partial\phi}U\right]r_{\text{ff}i} - \left[\frac{1}{r_{\text{ff}i}^2 + r_{\text{ff}j}^2}\frac{\partial}{\partial\lambda}U\right]r_{\text{ff}j}\right\}i$$
$$+\left\{\left[\frac{1}{r}\frac{\partial}{\partial r}U + \frac{r_{\text{ff}k}}{r^2\sqrt{r_{\text{ff}i}^2 + r_{\text{ff}j}^2}}\frac{\partial}{\partial\phi}U\right]r_{\text{ff}j} + \left[\frac{1}{r_{\text{ff}i}^2 + r_{\text{ff}j}^2}\frac{\partial}{\partial\lambda}U\right]r_{\text{ff}i}\right\}j$$
$$+\left\{\frac{1}{r}\left(\frac{\partial}{\partial r}U\right)r_{\text{ff}k} + \frac{\sqrt{r_{\text{ff}i}^2 + r_{\text{ff}j}^2}}{r^2}\frac{\partial}{\partial\phi}U\right\}k$$

given the following partial derivatives in spherical coordinates:

$$\frac{\partial}{\partial r}U = -\frac{\mu}{r^2}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\text{cb}}}{r}\right)^l (l+1)P_{l,m}[\sin(\phi)]\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\}$$
$$\frac{\partial}{\partial\phi}U = \frac{\mu}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\text{cb}}}{r}\right)^l \{P_{l,m+1}[\sin(\phi)] - (m)\tan(\phi)P_{l,m}[\sin(\phi)]\}\{C_{l,m}\cos(m\lambda) + S_{l,m}\sin(m\lambda)\}$$
$$\frac{\partial}{\partial\lambda}U = \frac{\mu}{r}\sum_{l=2}^{l_{\max}}\sum_{m=0}^{l}\left(\frac{R_{\text{cb}}}{r}\right)^l (m)P_{l,m}[\sin(\phi)]\{S_{l,m}\cos(m\lambda) - C_{l,m}\sin(m\lambda)\}$$

$P_{l,m}$ are associated Legendre functions.

$C_{l,m}$ and $S_{l,m}$ are the unnormalized harmonic coefficients.

Atmospheric Drag

The Orbit Propagator block supports inclusion of acceleration due to atmospheric drag using the NRLMSISE-00 atmosphere model. Atmospheric drag is a dominant perturbation in LEO and causes the ISS orbit to degrade over time without assistance from orbital maneuvers.

Atmospheric drag is calculated as:

$$\vec{a}_{\text{drag}} = -\frac{1}{2}\rho\left(\frac{C_D A_D}{m}\right)v_{\text{rel}}^2\frac{\vec{v}_{\text{rel}}}{v_{\text{rel}}}$$

where:

ρ is the atmospheric density.

C_D is the spacecraft drag coefficient.

A_D is the spacecraft drag area, or the area normal to v_{rel}.

m is the spacecraft mass.

$\vec{v}_{\text{rel}} = \vec{v}_{\text{icrf}} - \Omega \times \vec{r}_{\text{icrf}}$ is the spacecraft velocity relative to the atmosphere.

The NRLMSISE-00 atmospheric model is used to calculate atmospheric density. It requires space weather data (F10.7 and F10.7A radio flux values and geomagnetic indices), which can be obtained in a consolidated format from CelesTrak®. For more information about space weather data in MATLAB and Simulink®, see the Solar Flux and Geomagnetic Index block reference page and the Aerospace Toolbox `aeroReadSpaceWeatherData` function reference page.

Third Body Gravity

Third body gravity can be included in the Orbit Propagator block for the Moon, Sun, all planets in the solar system, or any arbitrary number of custom bodies. Gravity for each body is calculated as (shown for Sun):

$$\vec{a}_{\odot} = \mu_{\odot}\left(\frac{\vec{r}_{\mathrm{sat},\ \odot}}{{r_{\mathrm{sat},\ \odot}}^{3}} - \frac{\vec{r}_{\oplus,\ \odot}}{{r_{\oplus,\ \odot}}^{3}}\right)$$

where:

$\mu_{\odot}$ is the gravitational parameter of the Sun.

$\vec{r}_{\mathrm{sat},\ \odot}$ is the vector from the spacecraft to the center of the Sun, based on JPL DE405 planetary ephemeris data. For more information about planetary ephemerides, visit the Planetary Ephemeris block reference page or the Aerospace Toolbox `planetEphemeris` function reference page.

$\vec{r}_{\oplus,\ \odot}$ is the vector from the center of Earth to the center the Sun, based on JPL DE405 ephemeris data.

Solar Radiation Pressure

Solar radiation pressure has a minimal impact on orbit propagation in LEO. However, this example includes these calculations for completeness. The Orbit Propagator block calculates solar radiation pressure as:

$$\vec{a}_{\mathrm{srp}} = -\nu\, C_r \frac{A_s}{m}\, p_{\mathrm{srp}}\left(\frac{\mathrm{AU}}{r_{\mathrm{sat},\ \odot}}\right)^{2}\frac{\vec{r}_{\mathrm{sat},\ \odot}}{r_{\mathrm{sat},\ \odot}}$$

where:

ν is the shadow fraction. The value can be one of:

- Equal to 0 in umbra
- Between 0 and 1 in penumbra or antumbra
- Equal to 1 (full sunlight)

Depending on the mission requirements, two eclipse models with different levels of fidelity are supported on the block:

- The Dual cone shadow model calculates the fraction of the solar disk that is visible from the spacecraft position. It differentiates between partial, annular, and total eclipse, which means the spacecraft can be in sunlight, penumbra, antumbra, or umbra. The eclipse is partial in penumbra, annular in antumbra, and total in umbra.
- The Cylindrical shadow model assumes the Sun is infinitely far from the occulting bodies and the spacecraft, and that all rays of sunlight are parallel. In this case, the spacecraft is either in full sunlight or in umbra. The shadow fraction can be only 0 or 1.

C_r is the reflectivity coefficient.

A_s is the spacecraft SRP area, or the cross-sectional area seen by the Sun.

m is the spacecraft mass.

$p_{sr} = \frac{1353}{3e8} \frac{W/m^2}{m/s} = 4.5344321e^{-6} \frac{N}{m^2}$ is the solar radiation pressure at a distance of 1AU from the Sun.

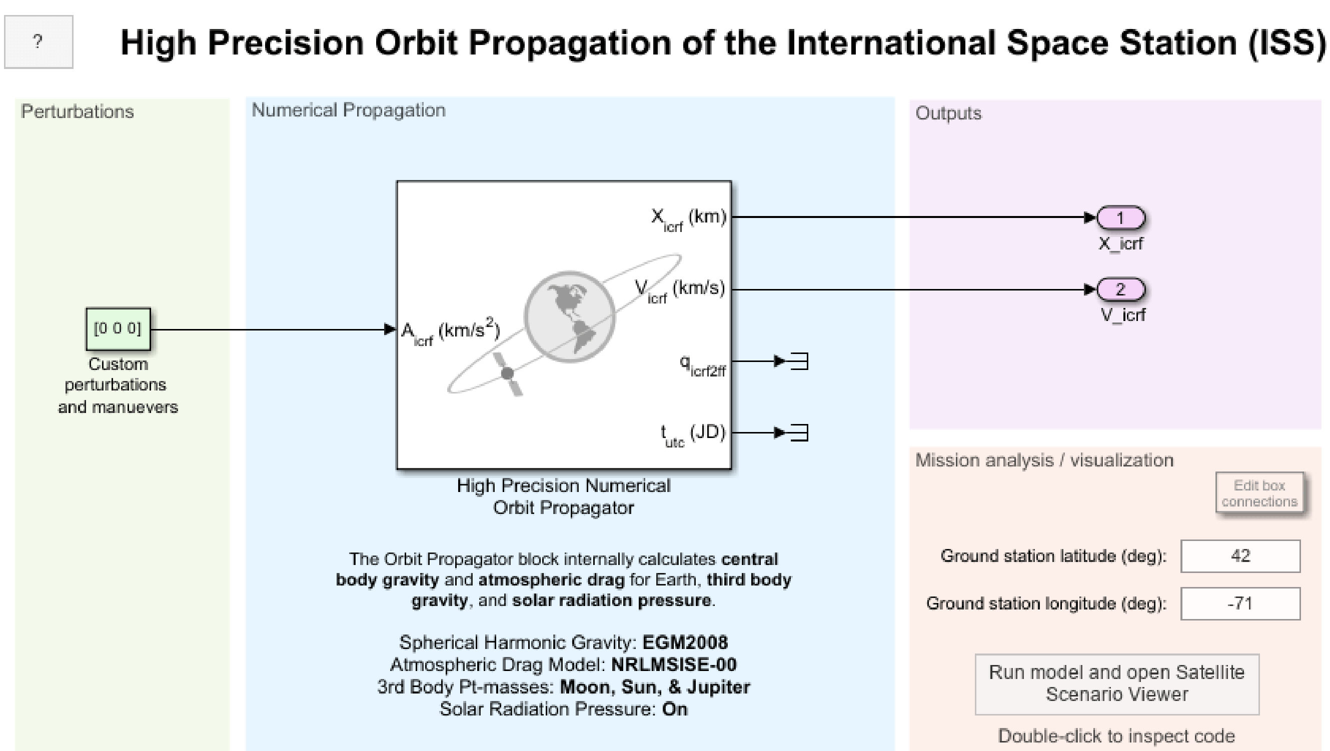
AU is the mean distance from the Sun to Earth (equal to 1 AU)

$\vec{r}_{sat,\odot}$ is the vector from the spacecraft to the center the Sun, based on JPL ephemeris data.

Open the Orbit Propagation Model

Open ISSHighPrecisionOrbitPropagationExampleModel.slx. This model contains an Orbit Propagator block connected to root-level outport blocks. The Orbit Propagator block internally calculates all perturbing accelerations used in the example. You can include additional perturbations, maneuvers, or any other effects as inputs to the block. The model also includes a "Mission analysis/visualization" section that contains a dashboard **Callback** button. When clicked, this button runs the model, creates a new `satelliteScenario` object in the base workspace containing the spacecraft defined in the Orbit Propagator block, and opens a Satellite Scenario Viewer window. To view the source code for this action, double-click the callback button. The "Mission analysis/visualization" section is a standalone workflow to create a new `satelliteScenario` object and is not used as part of this example.

```
mission.mdl = "ISSHighPrecisionOrbitPropagationExampleModel";
open_system(mission.mdl);
```



Configure the Simulink Model

Define the path to the Orbit Propagator block in the model.

```
iss.blk = mission.mdl + "/High Precision Numerical Orbit Propagator";
```

Set spacecraft initial conditions in the Simulink property inspector, or programmatically using `set_param`.

```
set_param(iss.blk, ...
    startDate = string(juliandate(mission.StartDate)), ...
    stateFormatNum = "ICRF state vector", ...
    inertialPosition = mat2str(iss.X0_icrf), ...
    inertialVelocity = mat2str(iss.V0_icrf));
```

Configure the propagator to include the perturbing accelerations outlined in the *High Precision Orbit Propagation Algorithm* section above. First, set central body gravity properties.

```
set_param(iss.blk, ...
    gravityModel = "Spherical harmonics", ...
    earthSH = "EGM2008", ...
    useEOPs = "on");
```

Set spacecraft properties for atmoshperic drag, third body gravity, and solar radiation pressure.

```
set_param(iss.blk, ...
    useDrag = "on", ...
    atmosModel = "NRLMSISE-00", ...
    SpaceWeatherDataFile = "aeroSpaceWeatherData.mat", ...
    mass = string(iss.Mass), ...
    dragCoeff = string(iss.DragCoeff), ...
    dragArea = string(iss.DragArea));

set_param(iss.blk, ...
    ephemerisModel = "DE405", ...
    useThirdBodyGravity = "on", ...
    includeSunGravity = "on", ...
    includeMoonGravity = "on", ...
    includeJupiterGravity = "on");

set_param(iss.blk, ...
    useSRP = "on", ...
    shadowModel = "Dual cone", ...
    includeMoonOccultation = "on", ...
    reflectivityCoeff = string(iss.SRPCoeff), ...
    srpArea = string(iss.SRPArea));
```

Apply model-level solver setting and save model output port data as a dataset of timetable objects.

```
set_param(mission.mdl, ...
    SolverType = "Variable-step", ...
    SolverName = "VariableStepAuto", ...
    RelTol = "1e-12", ...
    AbsTol = "1e-12", ...
    StopTime = string(seconds(mission.Duration)), ...
    SaveOutput = "on", ...
    OutputSaveName = "yout", ...
    SaveFormat = "Dataset", ...
    DatasetSignalFormat = "timetable");
```

Run the Model and Collect Ephemerides

Simulate the model. The Orbit Propagator block is configured to output position and velocity states in the ICRF coordinate frame. The simulation might take several minutes to run due to the strict tolerance settings we applied to the model in the previous section (1e-12).

```
mission.SimOutput = sim(mission.mdl);
```

Extract position and velocity data from the model output data structure.

```
iss.EphPosICRF = mission.SimOutput.yout{1}.Values;
iss.EphVelICRF = mission.SimOutput.yout{2}.Values;
```

Set the start data from the mission in the `timetable` object to convert the Time row from duration to datetime.

```
iss.EphPosICRF.Properties.StartTime = mission.StartDate;
iss.EphPosICRF.Time.TimeZone = "UTC";
iss.EphVelICRF.Properties.StartTime = mission.StartDate;
iss.EphVelICRF.Time.TimeZone = "UTC";
```

Import Ephemerides into a `satelliteScenario` Object

Create a satellite scenario object. For this example, do not explicitly specify a start and stop time. This omission allows the satellite scenario to derive the scenario time range from the `timetable` data passed to the `satellite()` method.

```
scenario = satelliteScenario;
```

Add the spacecraft to the `satelliteScenario` as ICRF position and velocity `timetable` objects using the `satellite` method.

```
iss.EphPosICRF_m = iss.EphPosICRF;
iss.EphPosICRF_m.Data = iss.EphPosICRF.Data*1e3;
iss.EphVelICRF_mps = iss.EphVelICRF;
iss.EphVelICRF_mps.Data = iss.EphVelICRF.Data*1e3;
iss.obj = satellite(scenario, iss.EphPosICRF_m, iss.EphVelICRF_mps, ...
    CoordinateFrame="inertial", Name="ISS");
```

Update the marker color to a darker shade to provide contrast against the trajectory lines.

```
iss.obj.MarkerColor = "#007F80";
iss.obj.MarkerSize = 10;
```

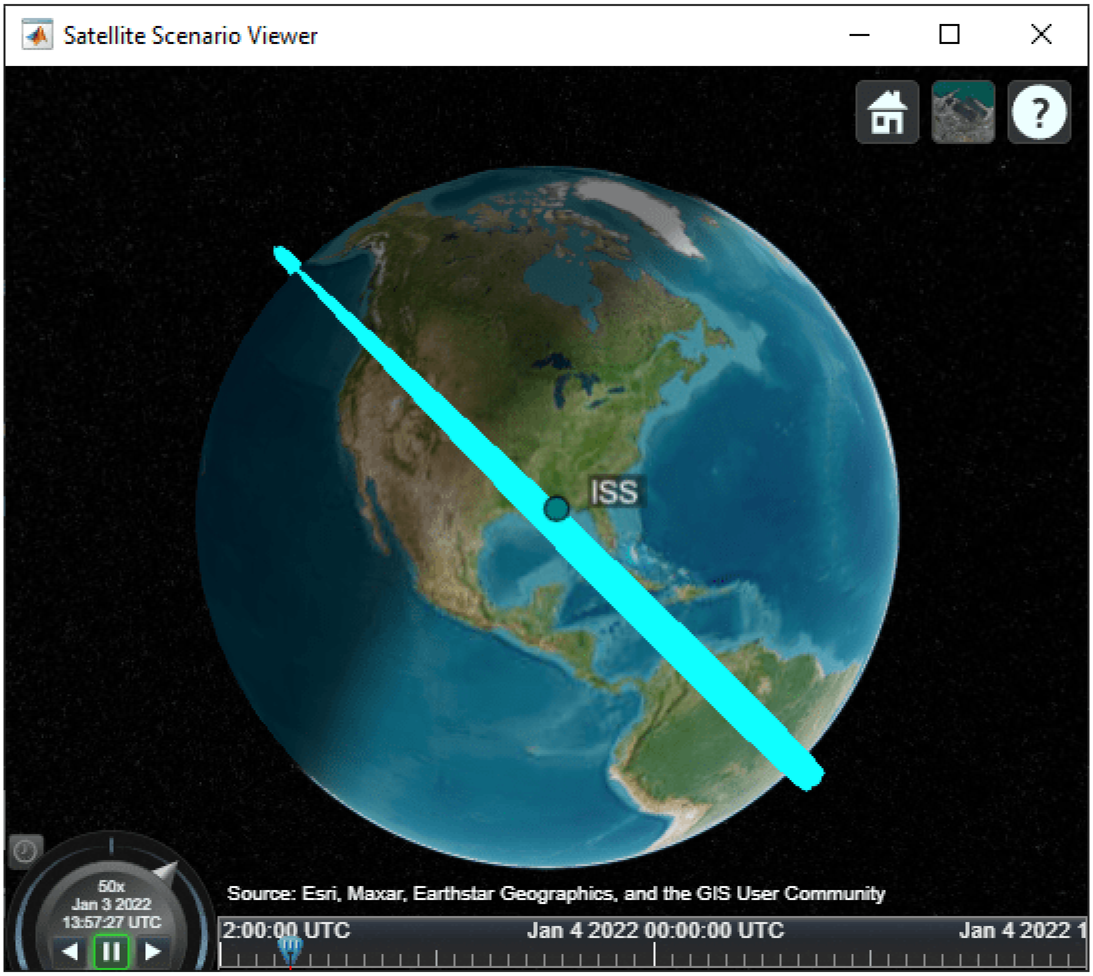
The start and stop time of the scenario are automatically adjusted to reflect the start and stop time of the `timetable` objects.

```
disp("Start: " + string(scenario.StartTime)); disp("Stop:  " + string(scenario.StopTime));
```

```
Start: 03-Jan-2022 12:00:00
Stop:  04-Jan-2022 12:00:00
```

Open the Satellite Scenario Viewer.

```
satelliteScenarioViewer(scenario);
```



Load the Reference Trajectory and Compare to the Calculated Trajectory

Load ISS trajectory data from Reference [1].

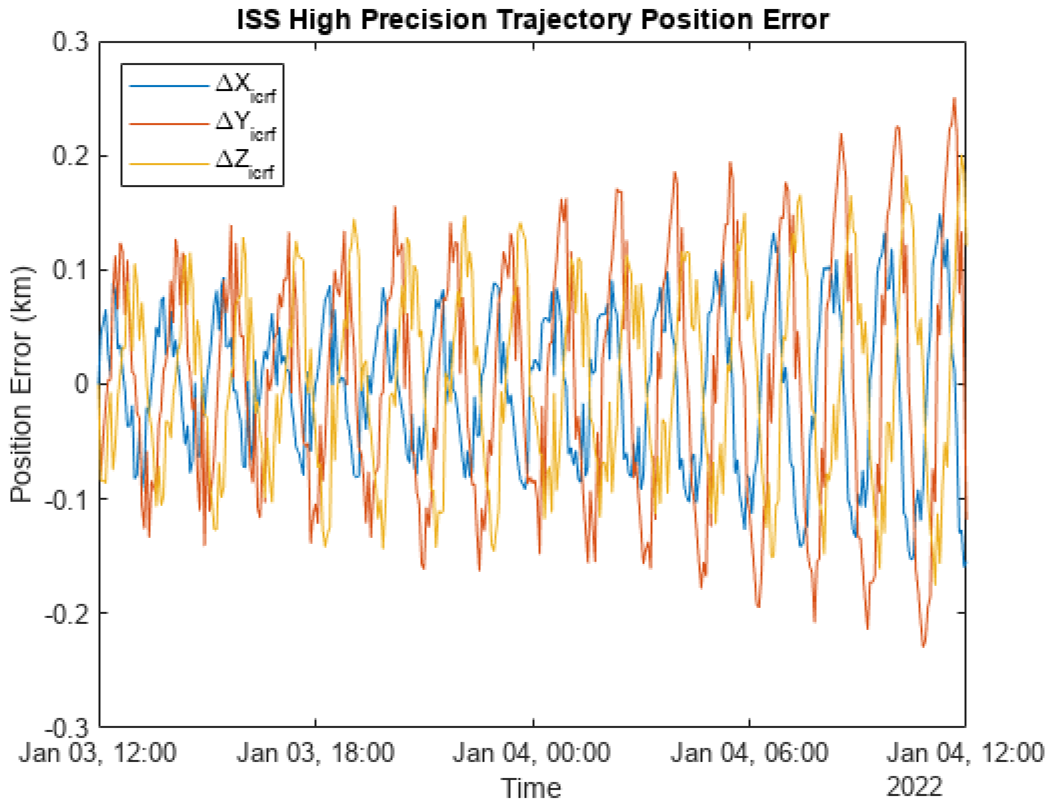
```
iss.RefEphemeris = load("ISS_ephemeris.mat", "pos_eme2000", "vel_eme2000");
```

Resample the simulation data to match the 4-minute sample rate of the reference trajectory.

```
iss.EphPosICRF = retime(iss.EphPosICRF, iss.RefEphemeris.pos_eme2000.Time);
iss.EphVelICRF = retime(iss.EphVelICRF, iss.RefEphemeris.vel_eme2000.Time);
```

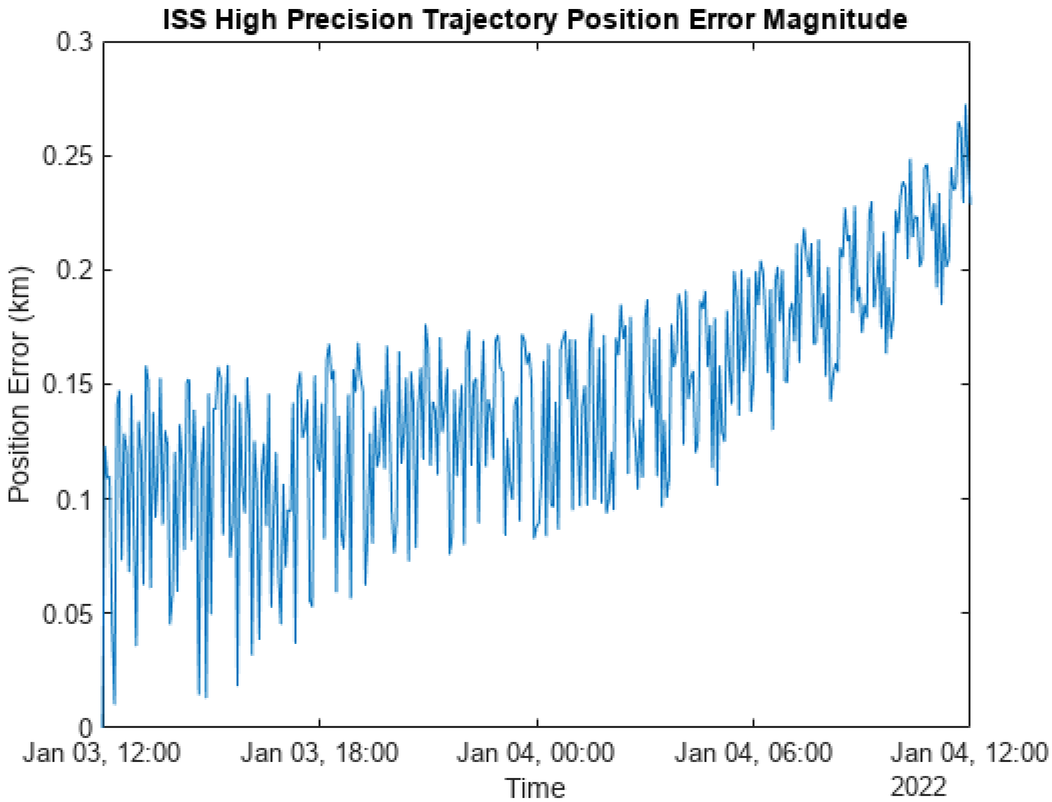
Plot the position error between the simulation data and the reference trajectory.

```
plot(iss.EphPosICRF.Time, iss.EphPosICRF.Data - iss.RefEphemeris.pos_eme2000.Position);
title("ISS High Precision Trajectory Position Error");
xlabel("Time");
ylabel("Position Error (km)");
legend(["\DeltaX_{icrf}","\DeltaY_{icrf}","\DeltaZ_{icrf}"], Location="northwest");
```



The maximum error between the simulation and the reference data over 24 hours is around 0.25km.

```
plot(iss.EphPosICRF.Time, vecnorm(iss.EphPosICRF.Data - iss.RefEphemeris.pos_eme2000.Position, 2
title("ISS High Precision Trajectory Position Error Magnitude");
xlabel("Time");
ylabel("Position Error (km)");
```



Compare High Precision Trajectory to Analytical Propagation

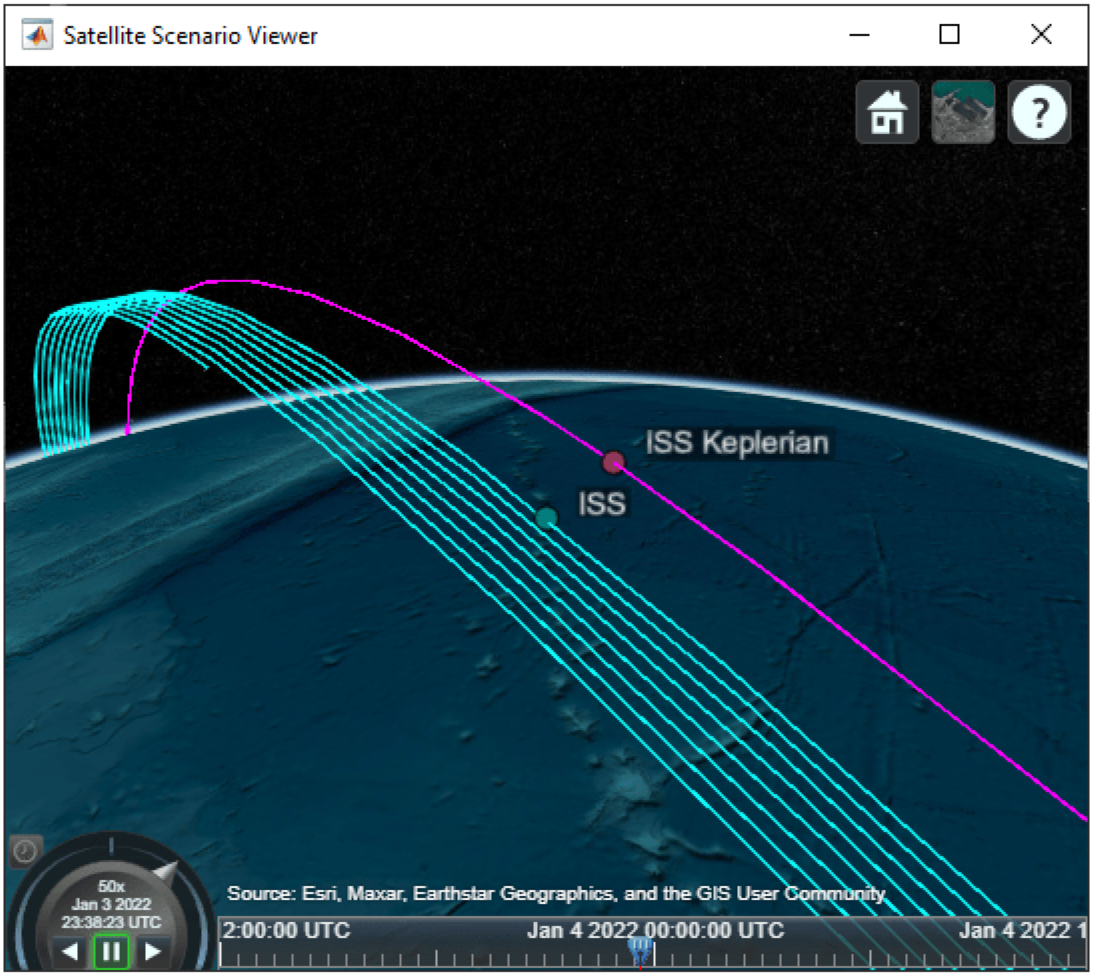
For comparison, add a new spacecraft to the satellite scenario using Two-Body-Keplerian propagation.

Convert initial position and velocity to orbital elements.

```
[iss.OrbEl.SemiMajorAxis, ...
    iss.OrbEl.Eccentricity, ...
    iss.OrbEl.Inclination, ...
    iss.OrbEl.RAAN, ...
    iss.OrbEl.ArgPeriapsis, ...
    iss.OrbEl.TrueAnomaly] =  ijk2keplerian(iss.X0_icrf*1e3, iss.V0_icrf*1e3);
```

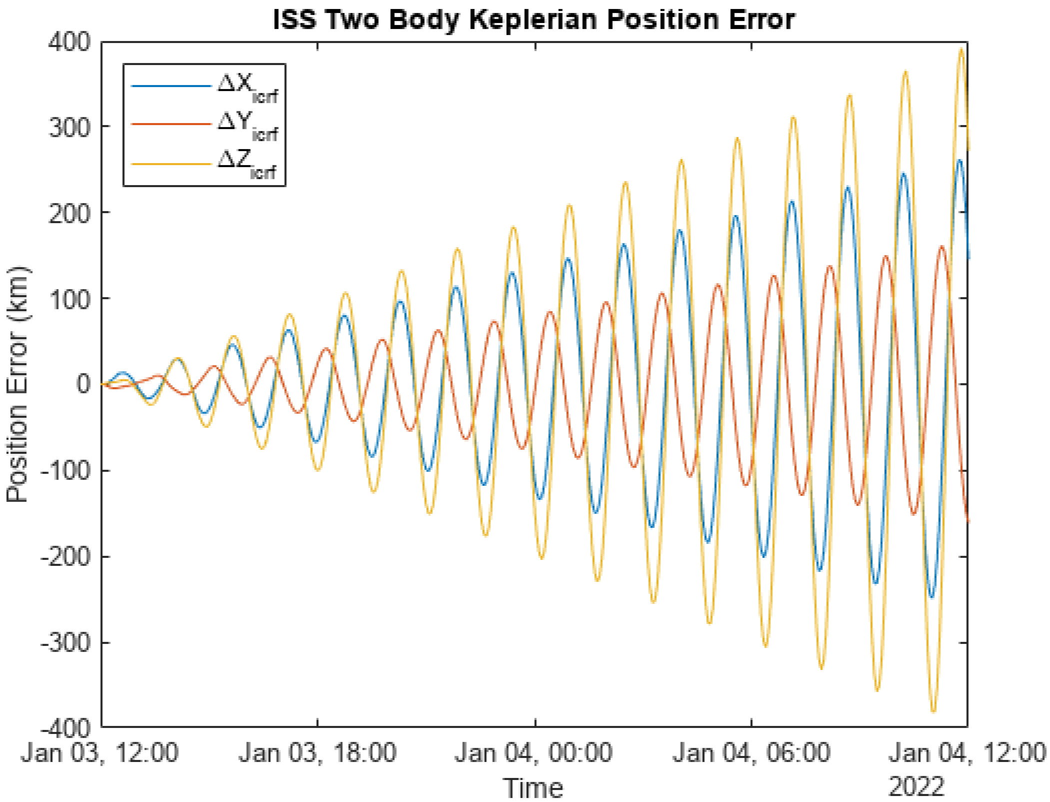
Add a new ISS object to the scenario using Two-Body-Keplerian propagation.

```
iss.KeplerianOrbit.obj = satellite(scenario, iss.OrbEl.SemiMajorAxis, ...
    iss.OrbEl.Eccentricity, iss.OrbEl.Inclination, iss.OrbEl.RAAN, ...
    iss.OrbEl.ArgPeriapsis, iss.OrbEl.TrueAnomaly, ...
    OrbitPropagator="two-body-keplerian", ...
    Name="ISS Keplerian");
iss.KeplerianOrbit.obj.Orbit.LineColor = "magenta";
iss.KeplerianOrbit.obj.MarkerColor = "#8E3A59";
iss.KeplerianOrbit.obj.MarkerSize = "10";
```



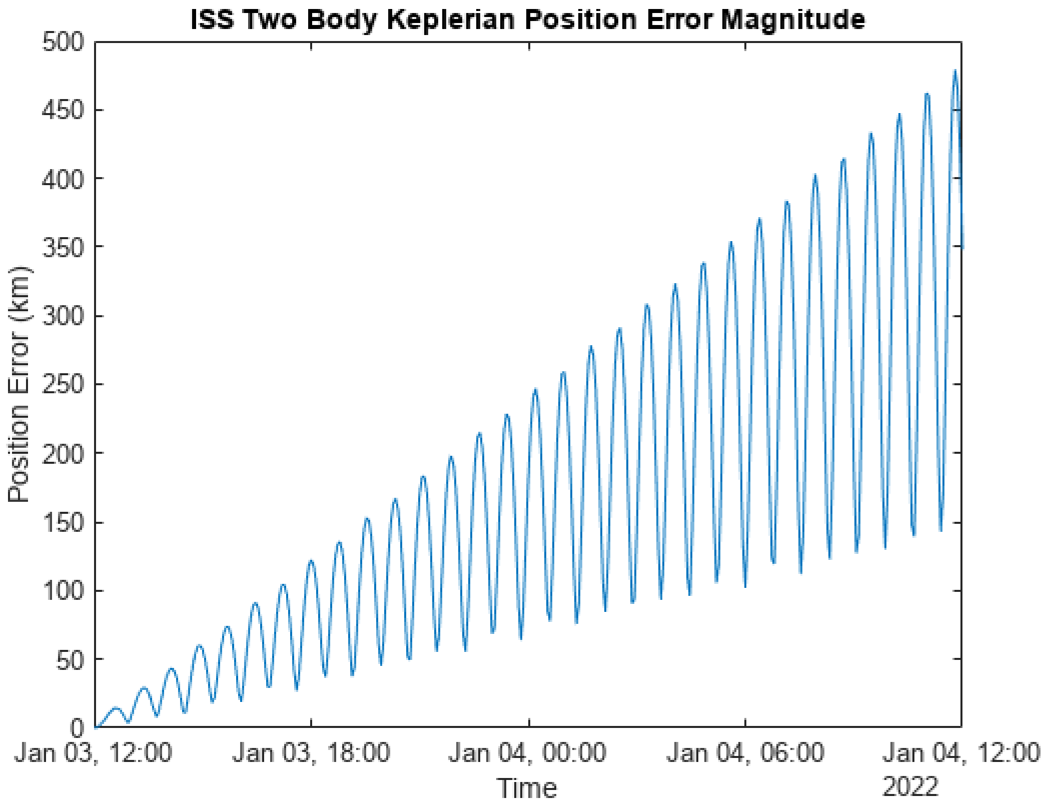
Plot the maximum error for Two-Body-Keplerian propagation.

```
[iss.KeplerianOrbit.EphPosICRFData, ~, ...
    iss.KeplerianOrbit.TimeSteps] = states(iss.KeplerianOrbit.obj, CoordinateFrame="inertial");
iss.KeplerianOrbit.EphPosICRF = retime(timetable(iss.KeplerianOrbit.TimeSteps', iss.KeplerianOrbi
    iss.RefEphemeris.pos_eme2000.Time);
plot(iss.EphPosICRF.Time, iss.KeplerianOrbit.EphPosICRF.Data/1e3 - iss.RefEphemeris.pos_eme2000.I
title("ISS Two Body Keplerian Position Error");
xlabel("Time");
ylabel("Position Error (km)");
legend(["\DeltaX_{icrf}","\DeltaY_{icrf}","\DeltaZ_{icrf}"], Location="northwest");
```



The maximum error between the Two-Body-Keplerian propagation and the reference data over 24 hours approaches exceeds 450km, compared to 0.25km for the high precision numerical propagation simulation.

```
plot(iss.EphPosICRF.Time, vecnorm(iss.KeplerianOrbit.EphPosICRF.Data/1e3 - iss.RefEphemeris.pos_e
title("ISS Two Body Keplerian Position Error Magnitude");
xlabel("Time");
ylabel("Position Error (km)");
```



References

[1] NASA Open Data Portal - Trajectory Operations and Planning (TOPO), ISS_COORDS_2022-01-03 Public Distribution File - https://data.nasa.gov/Space-Science/ISS_COORDS_2022-01-03/ffwr-gv9k

[2] Vallado, David. Fundamentals of Astrodynamics and Applications, 4th ed. Hawthorne, CA: Microcosm Press, 2013.

See Also

Blocks
Orbit Propagator | Spherical Harmonic Gravity Model | NRLMSISE-00 Atmosphere Model

Objects
`satelliteScenario`

Parachute Simulation Study with Monte Carlo Analysis

This example shows the simulation of a package with a parachute being dropped from an airplane. The parachute opens out of the package and induces a drag force, slowing the descent. The example uses Monte Carlo analysis to perform the sensitivity study of the parachute simulation with uncertainty in wind disturbance, airdrop altitude of the package from the aircraft, and opening altitude of the parachute.

Simulink Model Description

To understand the parachute simulation architecture, open the Simulink™ model.

```
open_system('ParachuteSim.slx');
```

The top model consists of two subsystems:

Environment defines the environment in which the parachute operates. It consists of the following blocks:

- Flat Earth to LLA block converts Earth position to latitude, longitude, and altitude. The initial heading angle and the initial latitude and longitude of the package drop are specified as parameters in the block. The variable *headingInitial* corresponds to the initial heading angle in degrees. The variables *latInitial* and *longInitial* correspond to the geodetic latitude and longitude in degrees.

```
headingInitial    = 45 ;
latInitial        = 43.5319 ;
longInitial       = 1.2438 ;
```

- COESA Atmosphere Model calculates the density value corresponding to the altitude.
- WGS84 Gravity Model computes the Earth gravity at a specific latitude, longitude, and altitude.
- A realistic wind environment is created using the combination of Wind Shear Model to calculate the mean wind speed, Dryden Wind Turbulence Model (Continuous) to add wind turbulence, and Discrete Wind Gust Model to generate gust wind.

Parachute System Model uses the 6DOF (Euler Angles) block to simulate the parachute model based on the initial conditions, weight, and drag force. When a package containing the parachute is dropped from an aircraft, two forces act on it, the weight of the package and the aerodynamic drag. The aerodynamic drag force of the package is negligible before the parachute opens, resulting in a projectile motion of the package. Once the parachute opens, the aerodynamic drag on the parachute slows the descent. The default values of the parachute are based on the EPC (Ensemble de Parachutage du Combattant), one of the French parachutes manufactured by IrvinGQ (formerly Airborne Systems).

As the aerodynamic drag and the weight of the parachute are opposing forces, the net vertical force is given by the difference between the weight (mass × g) and the drag force $\left(\frac{1}{2} \times \text{rho} \times \text{velocity}^2 \times C_d \times \text{area}\right)$, where:

- *mass* represents the mass of the parachute in kg

- g is the acceleration due to gravity approximated to 9.81 $\frac{m}{s^2}$
- *rho* is the density of air approximated to 1.229 $\frac{\text{kg}}{m^3}$
- *velocity* corresponds to the velocity of parachute in $\frac{m}{s}$
- C_d is the drag coefficient
- *area* represents the area of the parachute in m^2

During the descent, a point is quickly reached when the net force on the parachute is zero and the drag equals weight. This relation is used to compute the drag coefficient of the parachute, as follows:

```
area        = [115];
mass        = [165];
velocity    = [6];
g           = 9.81;
rho         = 1.229;
Cd          = 2*mass*g/(rho*(velocity^2)*area);
```

The gravity and density values corresponding to the altitude and wind velocity computed in the Environment subsystem are used with the drag coefficient, opening altitude, and mass of the parachute to obtain the net force acting on the parachute. The opening altitude, denoted by the variable *hOpening*, corresponds to the altitude (in meters) at which the parachute is deployed from the package. Observe that the opening altitude should be less than the airdrop altitude from an aircraft. The variable *packageMass* corresponds to the mass of the package with the parachute in kg. By default, the opening altitude and the package mass are set to 500 m and 130 kg, respectively.

```
hOpening    = [500];
packageMass = [130];
```

The net moments on the parachute are assumed zero. The net forces and moments are subsequently fed to the 6DOF (Euler Angles) block to simulate the parachute model based on the specified initial conditions in the block. The initial position in the inertial axis is specified below. The variable *ZeInitial* corresponds to the airdrop altitude (in meters) of the package from the aircraft. By default, the airdrop altitude of the package is set to 2000 m.

```
XeInitial          = [0];
YeInitial          = [0];
ZeInitial          = [2000];
```

The initial velocity of the package along the x-axis corresponds to the velocity of the aircraft when the package is dropped, and is given a default value of 235 km/h. The initial velocities of the package along y and z-axes are considered zero (in km/h). The initial Euler angle and Euler rates are considered zero. The initial velocities in the three body axes are converted to m/s using the `convvel` function as shown below:

```
UInitialKMPH       = [235];
UInitialMPS        = convvel(UInitialKMPH,'km/h','m/s');
```

```
VInitialKMPH       = 0 ;
VInitialMPS        = convvel(VInitialKMPH,'km/h','m/s');
WInitialKMPH       = 0 ;
WInitialMPS        = convvel(WInitialKMPH,'km/h','m/s');
```

Parachute Simulation

By default, the wind disturbance is introduced in the parachute simulation. The parameter **Gust start time** in Discrete Wind Gust Model corresponds to the time the gust begins. As the time taken for the parachute to reach the ground is around 100 seconds, the variable *windSeed* that corresponds to the **Gust start time** parameter is taken as a random integer within a range of 1 to 100 using the `randi` function. To simulate the parachute model without wind disturbance, set the w*indDisturbance* variable to `false`. This switching is carried out in the Simulink model using a Variant Subsystem, Variant Model, Variant Assembly Subsystem with *windDisturbance* as the variant choice variable.

```
windDisturbance = Yes ;
if windDisturbance == true
    windSeed = randi(100,1);
end
```

After initializing and setting up all the parameters of the parachute simulation model, the `sim` function simulates it until the parachute touches the ground. The results are stored in the variable *SimResults* as a `Simulink.SimulationOutput` object.

```
simResults = sim('ParachuteSim.slx');
```

The simulation results characterize the response of the parachute. An overview is given by the state of the parachute at the touchdown, as follows.

The time taken for the parachute to hit the ground in seconds:

```
TFinal          = simResults.tout(end)
```

```
TFinal = 108.3395
```

The position along the *x*-axis where the parachute hits the ground in meters:

```
XeFinal          = simResults.yout{1}.Values.Data(end,1)
```

```
XeFinal = 1.1935e+03
```

The position along the *y*-axis where the parachute hits the ground in meters:

```
YeFinal          = simResults.yout{1}.Values.Data(end,2)
```

```
YeFinal = -164.5865
```

The velocity along the *x*-axis with which the parachute hits the ground in m/s:

```
UeFinal          = simResults.yout{2}.Values.Data(end,1)
```

```
UeFinal = -1.9773
```

The velocity along the *y*-axis with which the parachute hits the ground in m/s:

```
VeFinal          = simResults.yout{2}.Values.Data(end,2)
```

```
VeFinal = -2.8996
```

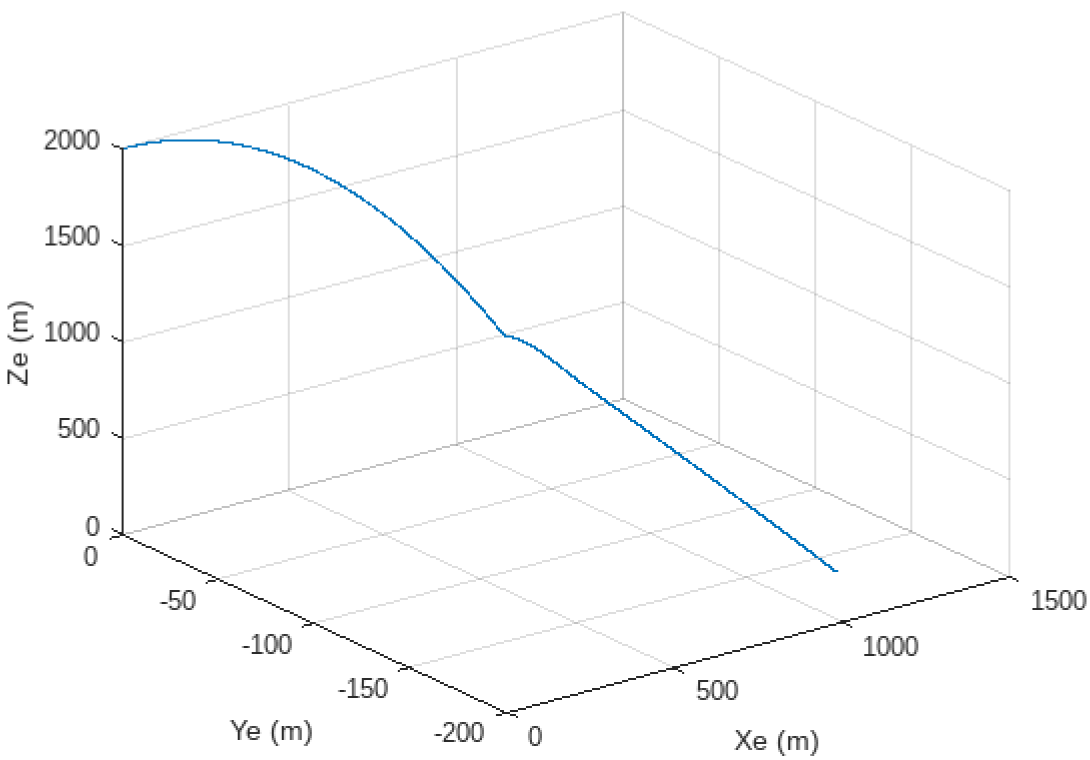
The velocity along the *z*-axis with which the parachute hits the ground in m/s:

```
WeFinal          = simResults.yout{2}.Values.Data(end,3)
```

```
WeFinal = -5.1909
```

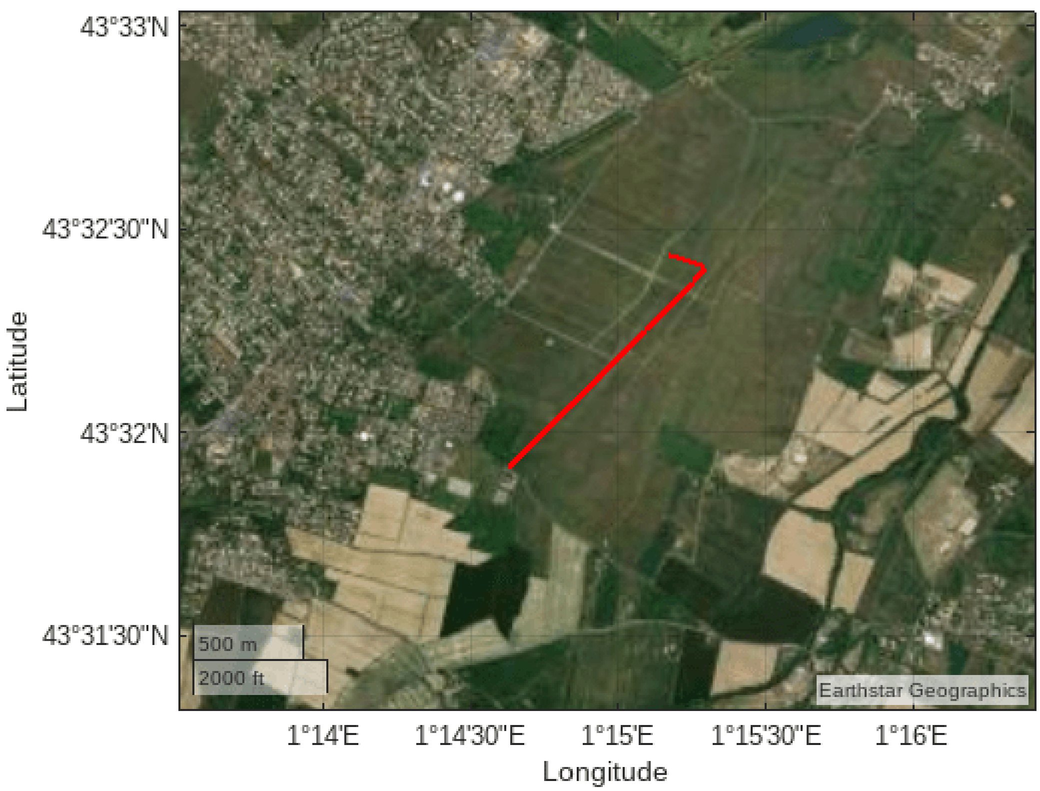
The variables *YeFinal* and *VeFinal* depend on the wind disturbance. If an ideal scenario with no wind condition is considered, then these variables are zero. This assumption results in only longitudinal motion of the parachute, represented in 2D with *x* and *z* coordinates. However, in a realistic environment, the package exhibits both lateral and longitudinal motion, which is plotted below in 3D. From the plot, observe that the package exhibits a projectile motion until the parachute opens and induces an aerodynamic drag. This drag considerably slows down the descent.

```
figure;
plot3(simResults.yout{1}.Values.Data(:,1),simResults.yout{1}.Values.Data(:,2),...
      simResults.yout{1}.Values.Data(:,3));
zlim([0 max(simResults.yout{1}.Values.Data(:,3))]);
grid on;
xlabel('Xe (m)');
ylabel('Ye (m)');
zlabel('Ze (m)');
```



The planar satellite view of the parachute simulation is depicted below. The `flat2lla` function estimates latitude, longitude, and altitude from the Earth position coordinates. The `geoplot` function plots the resulting geographic coordinates. The `geobasemap` function changes the basemap to `satellite`.

```
figure;
llaPosition = flat2lla([simResults.yout{1}.Values.Data(:,1) ...
                        simResults.yout{1}.Values.Data(:,2) ...
                     -1*simResults.yout{1}.Values.Data(:,3)],...
                     [latInitial, longInitial], headingInitial,0);
geoplot(llaPosition(:,1), llaPosition(:,2),'r','LineWidth',2)
geolimits([min(llaPosition(:,1))-0.01 max(llaPosition(:,1))+0.01],...
          [min(llaPosition(:,2))-0.01 max(llaPosition(:,2))+0.01])
geobasemap satellite
```



Monte Carlo Analysis Using Parallel Simulation

The Monte Carlo simulations assess the response of the parachute under uncertain conditions. The variable *numSim* provides the number of simulations in the Monte Carlo analysis. In the parachute simulation, the uncertainty in wind disturbance, airdrop altitude of the package, and the opening altitude of the parachute are considered. The randomness in wind disturbance is accommodated by varying the start time of the gust in the Discrete Wind Gust Model block within the range of 1 to 100.

```
numSim              = 100;
windDisturbance     = true;
gustStartTime       = randi(100,[1 numSim]);
```

The uncertainty in the airdrop altitude of the package and the opening altitude of the parachute is assumed to follow a normal distribution curve with the mean set at the initial conditions of *ZeInitial* and *hOpening*, respectively. The standard deviations of the airdrop altitude and the opening altitude are specified below:

```
ZeDeviation        = 5;
hOpeningDeviation  = 5;
```

The random values of the airdrop altitude and the opening altitude are then drawn from a normal distribution with the specified mean and standard deviation using the `randn` function.

```
airdropAltitude = ZeDeviation.*randn(numSim,1) + ZeInitial;
openingAltitude = hOpeningDeviation.*randn(numSim,1) + hOpening;
```

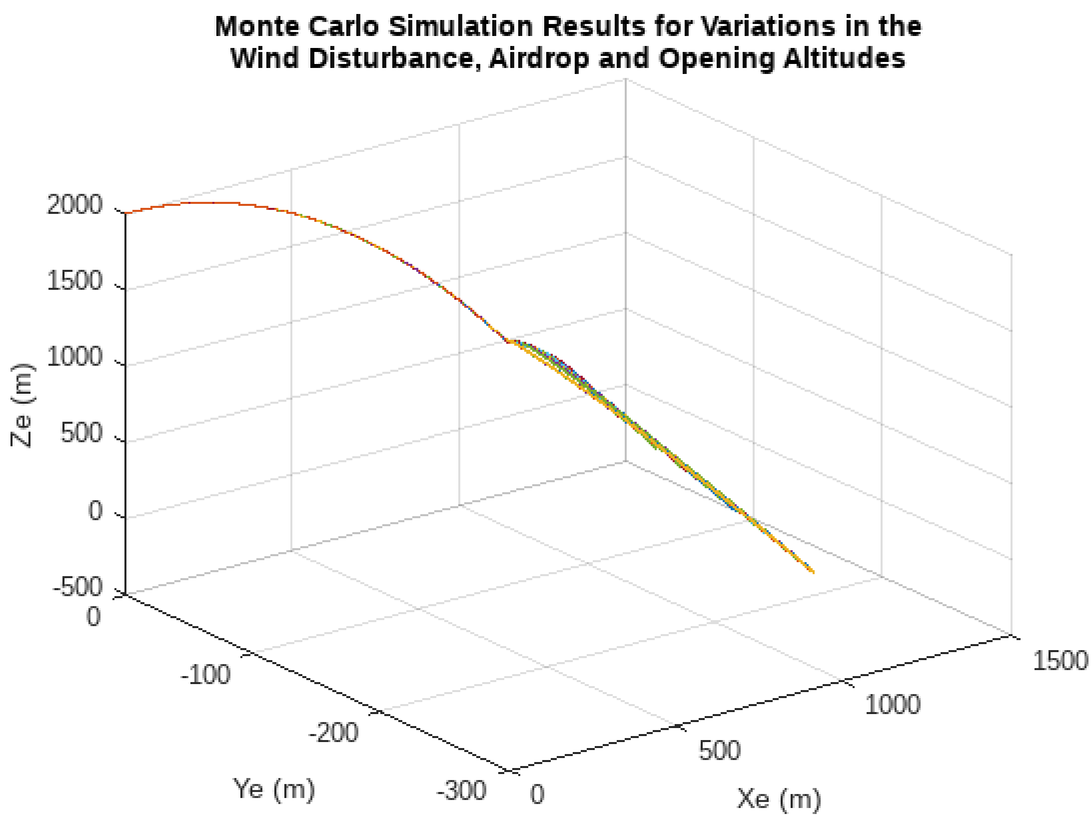
The Monte Carlo analysis requires simulating the parachute system multiple times, so this example uses the `parsim` function. The `Simulink.SimulationInput` object specifies the varying inputs of the simulations. The `parsim` function is configured to run simulations using fast restart, to monitor progress using Simulation Manager, to track the progress of simulations, and to transfer the variables in the base workspace to the parallel workers using Parallel Computing Toolbox™.

```
in(1:numSim) = Simulink.SimulationInput('ParachuteSim');
for simNumber = 1:1:numSim
    in(simNumber) = in(simNumber).setVariable('hOpening',openingAltitude(simNumber));
    in(simNumber) = in(simNumber).setVariable('Ze',airdropAltitude(simNumber));
    in(simNumber) = in(simNumber).setVariable('windSeed',gustStartTime(simNumber));
end
simResults = parsim(in, 'UseFastRestart','on','ShowSimulationManager', 'on',...
                    'ShowProgress','on','TransferBaseWorkspaceVariables','on');
```

```
[11-Apr-2023 19:39:36] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 8 workers.
[11-Apr-2023 19:40:30] Starting Simulink on parallel workers...
[11-Apr-2023 19:42:09] Configuring simulation cache folder on parallel workers...
[11-Apr-2023 19:42:10] Transferring base workspace variables used in the model to parallel worke
[11-Apr-2023 19:42:10] Loading model on parallel workers...
[11-Apr-2023 19:43:32] Running simulations...
[11-Apr-2023 19:45:06] Cleaning up parallel workers...
```

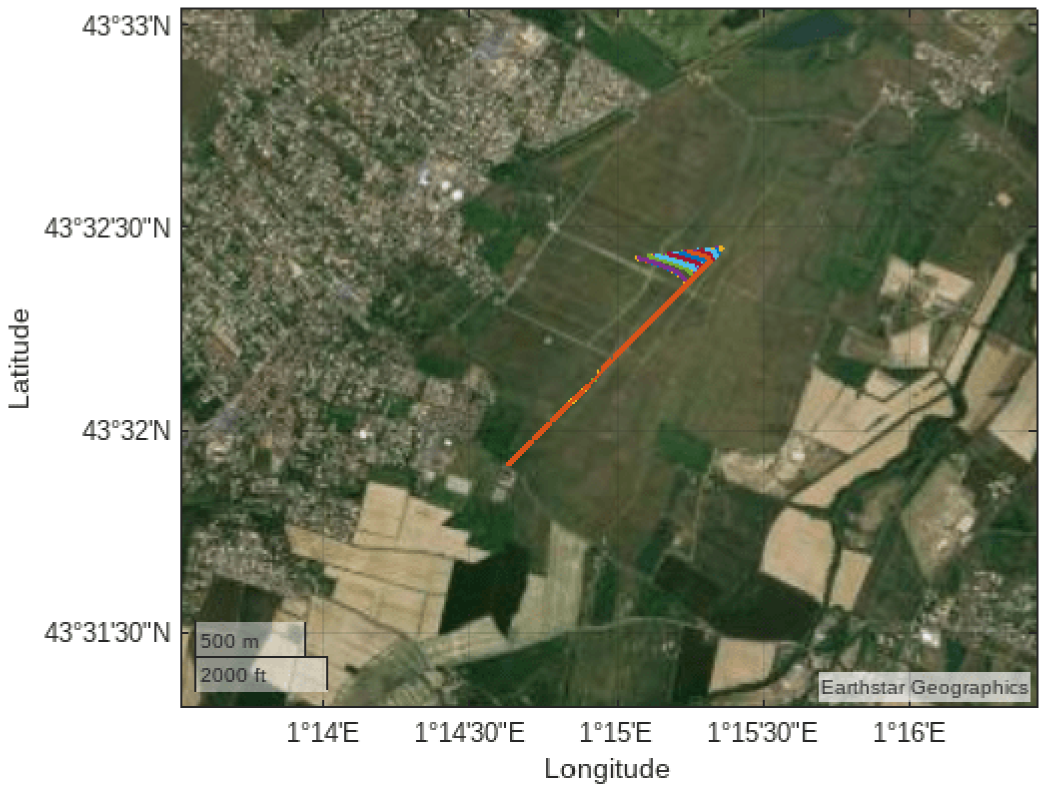
The results of the multiple simulations are plotted below. From the plot, it is seen that the input uncertainties result in a small deviation of the landing site of the parachute.

```
figure;
for simNumber = 1:1:numSim
    plot3(simResults(1,simNumber).yout{1}.Values.Data(:,1),...
          simResults(1,simNumber).yout{1}.Values.Data(:,2),...
          simResults(1,simNumber).yout{1}.Values.Data(:,3));
    hold on;
end
grid on;
title(["Monte Carlo Simulation Results for Variations in the";...
       "Wind Disturbance, Airdrop and Opening Altitudes"]);
xlabel('Xe (m)');
ylabel('Ye (m)');
zlabel('Ze (m)');
```



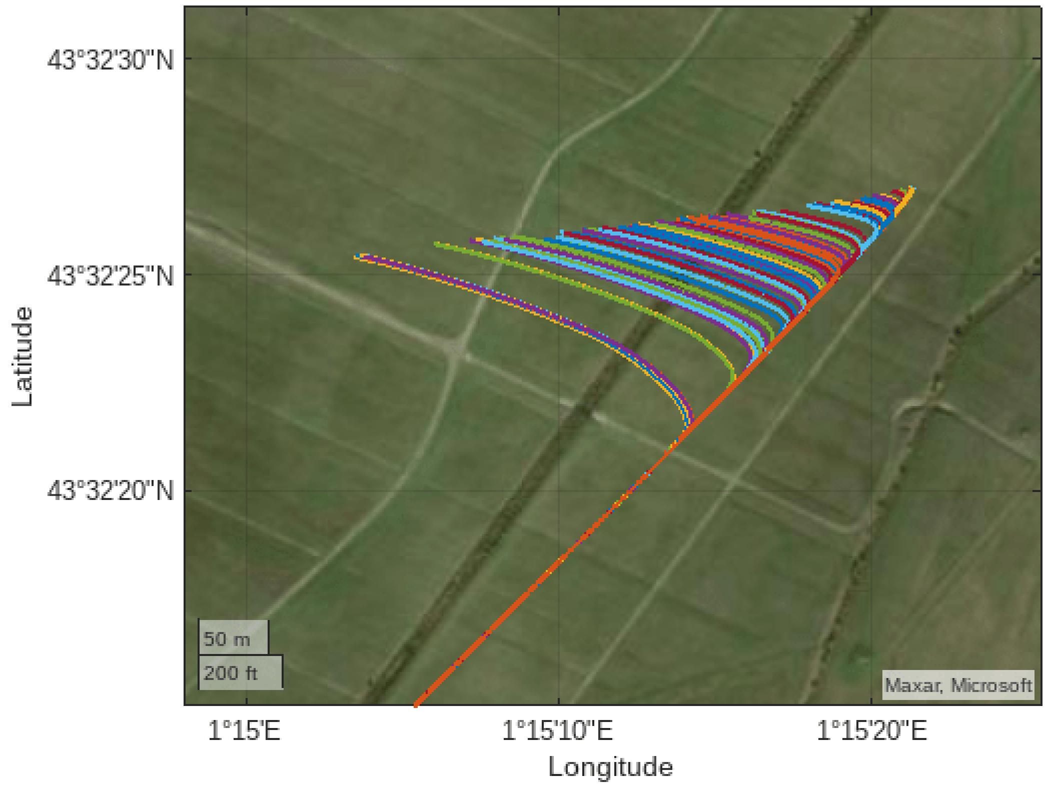
The simulation results are also represented as a geoplot in the satellite view.

```
figure;
gx = geoaxes;
for simNumber = 1:1:numSim
    llaPosition = flat2lla([simResults(1,simNumber).yout{1}.Values.Data(:,1) ...
                            simResults(1,simNumber).yout{1}.Values.Data(:,2) ...
                           -1*simResults(1,simNumber).yout{1}.Values.Data(:,3)],...
                           [latInitial, longInitial], headingInitial,0);
    geoplot(gx,llaPosition(:,1), llaPosition(:,2),'LineWidth',2)
    geolimits(gx,[min(llaPosition(:,1,:))-0.01 max(llaPosition(:,1,:))+0.01],...
                 [min(llaPosition(:,2,:))-0.01 max(llaPosition(:,2,:))+0.01])
    geobasemap(gx,"satellite")
    hold on
end
```



The plot is zoomed in to reflect the differences in the landing sites corresponding to each of the parachute simulations.

```
figure;
gx = geoaxes;
for simNumber = 1:1:numSim
    llaPosition = flat2lla([simResults(1,simNumber).yout{1}.Values.Data(:,1) ...
                            simResults(1,simNumber).yout{1}.Values.Data(:,2) ...
                         -1*simResults(1,simNumber).yout{1}.Values.Data(:,3)],...
                           [latInitial, longInitial], headingInitial,0);
    geoplot(gx,llaPosition(:,1), llaPosition(:,2),'LineWidth',2)
    geolimits(gx,[43.5375 43.5420],[1.2495 1.2570])
    geobasemap(gx,"satellite")
    hold on
end
```



The example demonstrates Monte Carlo analysis for parachute simulation with variations in the wind disturbance, airdrop and opening altitudes. Observe the simulation results of Monte Carlo analysis by varying the start time of the wind gust and standard deviations of the airdrop and opening altitudes.

Delete Current Pool

Use the parallel pool object to delete the current pool of parallel workers created by `parsim`.

```
delete(gcp('nocreate'));
```

```
Parallel pool using the 'Processes' profile is shutting down.
```

Reference

https://defense-militaire.over-blog.com/2020/12/combien-coutent-les-ensembles-de-parachutage-du-combattant-epc.html

Hohmann Transfer with the Spacecraft Dynamics Block

This example shows how to model a Hohmann transfer of a spacecraft between two circular coplanar orbits in Simulink® and visualize the transfer in a satellite scenario. It uses:

- Aerospace Blockset™ **Spacecraft Dynamics** block
- Aerospace Blockset **Attitude Profile** block
- Aerospace Toolbox **`satelliteScenario`** object

The Aerospace Blockset **Spacecraft Dynamics** block models translational and rotational dynamics of spacecraft using numerical integration. To represent thrust from the propulsion system and the moments generated by the attitude controller, you can configure the block to accept propellant mass flow rate, exhaust velocity, and body moments. To implement the Hohmann transfer, perform the necessary maneuvers by applying the necessary moments to orient the spacecraft in the desired direction and applying thrust to achieve the necessary delta-v.

The attitude controller consists of the Aerospace Blockset **Attitude Profile** block, which returns the shortest quaternion rotation that aligns the spacecraft's provided alignment axis with the specified target. This quaternion constitutes the alignment error, which is corrected by a cascade controller. The inner loop of the controller controls the angular velocity. The outer loop of the controller controls the attitude. Both loops involve a simple proportional controller. In this example, the spacecraft's x-axis (assumed to be the thrust axis) is commanded to align with the desired delta-v direction.

The Aerospace Toolbox **`satelliteScenario`** object lets you load previously generated, time-stamped ephemeris and attitude data into a scenario from a timeseries or timetable object. This data is interpolated in the scenario object to align with the scenario time steps, allowing you to incorporate data generated in a Simulink model into either a new or existing **`satelliteScenario`** object. In this example, the **Spacecraft Dynamics** block computes the satellite orbit and attitude states. For visualization on the satellite scenario viewer, this data is then used to add a satellite to a new **`satelliteScenario`** object.

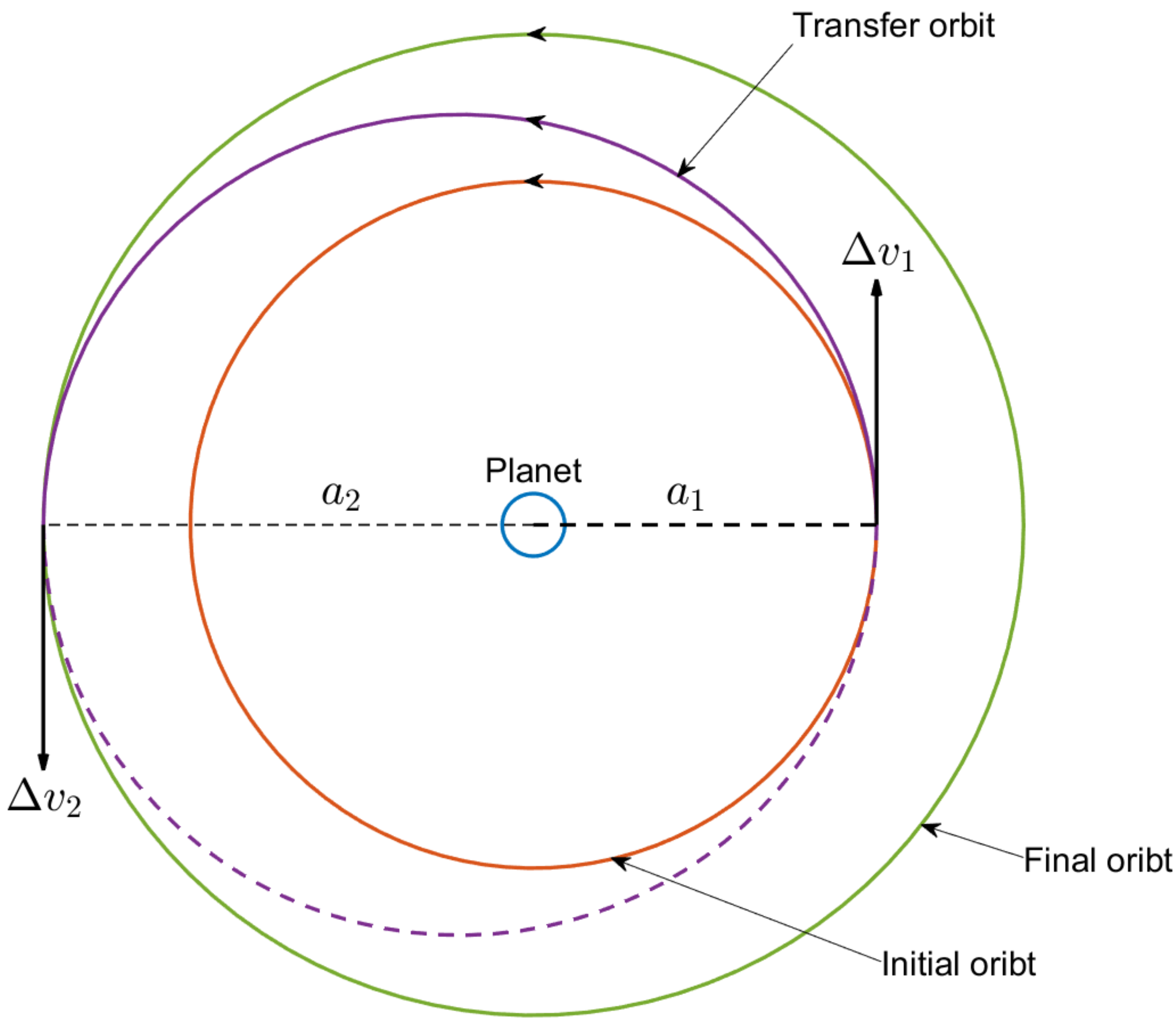
The Hohmann Transfer

The Hohmann transfer consists of two coplanar maneuvers to transfer a spacecraft between two orbits that lie on the same plane. The initial and final orbits can be circular or elliptical. In this example, the orbits are assumed to be circular. The transfer orbit is an ellipse tangential to the two orbits. The transfer involves two maneuvers:

- The first maneuver at the tangent point of the initial orbit (the first transfer point) places the spacecraft on the elliptic transfer orbit.
- The second maneuver at the tangent point of the final orbit (the second transfer point) inserts the spacecraft into this orbit.

The transfer angle, the angle between the position vector of the spacecraft at the two transfer points with respect to the planet, is 180 degrees. Because the transfer ellipse is tangential, the required velocity change (delta-v) at the transfer points is the difference between the circular orbital speed and the velocity on the elliptic transfer orbit at the transfer points. The direction of the delta-v is the tangent at the transfer points.

The following figure illustrates a Hohmann transfer between two coplanar circular orbits. If the second maneuver is not performed, the spacecraft follows the dashed magenta back to the first tangent point.



Suppose a_1 and a_2 are the orbital radii of the initial and final circular orbits. The semimajor axis of the transfer orbit is given by:

$$a_T = \frac{(a_1 + a_2)}{2}$$

The velocity magnitude at any given point on the initial circular orbit is given by:

$$v_1 = \sqrt{\frac{\mu}{a_1}}$$

where μ is the standard gravitational parameter of the planet (Earth). The velocity on the transfer orbit at the first transfer point is derived from the following energy equation:

$$\frac{v_{T,1}^2}{2} - \frac{\mu}{a_1} = -\frac{\mu}{2a_T}$$

where $v_{T,1}$ is the velocity on the transfer orbit at the first transfer point (the tangent between the initial orbit and the transfer orbit). Therefore:

$$v_{T,1} = \sqrt{2\mu\left(\frac{1}{a_1} - \frac{1}{2a_T}\right)}$$

Consequently, the delta-v at the first transfer point (Δv_1) is given by:

$$\Delta v_1 = v_{T,1} - v_1$$

The direction of Δv_1 is the tangent at the first transfer point on the initial orbit, as illustrated above. Similarly, the delta-v at the second transfer point (Δv_2) can be calculated as follows:

$$v_2 = \sqrt{\frac{\mu}{a_2}}$$

$$v_{T,2} = \sqrt{2\mu\left(\frac{1}{a_2} - \frac{1}{2a_T}\right)}$$

$$\Delta v_2 = v_2 - v_{T,2}$$

As before, the direction of Δv_2 is the tangent at the second transfer point on the final orbit. The time taken to coast from the first transfer point to the second (transfer duration) is half the orbital period of the transfer orbit.

$$t_T = \pi\sqrt{\frac{a_T^3}{\mu}}$$

Mission Overview

The mission begins on December 14, 2022, 1:04 AM UTC. Define this time as a datetime object and a Julian date.

```
initialTime = datetime(2022,12,14,1,4,0);
initialTimeJD = juliandate(initialTime);
```

At the initial time, the spacecraft is in a circular orbit with the following specifications:

- Semimajor axis (a_1) = 7,000 km
- Eccentricity = 0
- Inclination = 45 degrees
- Right ascension of ascending node = 90 degrees
- Argument of periapsis = 30 degrees
- True anomaly = 0 degrees

Set the semimajor axis, `a1`, of the initial orbit in meters.

```
a1 = 7000e3;
```

Set the eccentricity, `e1`, of the initial orbit.

```
e1 = 0;
```

Set the inclination, `i1`, of the initial orbit in degrees.

```
i1 = 45;
```

Set the right ascension of ascending node, `raan1`, of the initial orbit in degrees.

```
raan1 = 90;
```

Set the argument of periapsis, `w1`, of the initial orbit in degrees.

```
w1 = 30;
```

Set the true anomaly, `theta1`, of the initial orbit in degrees.

```
theta1 = 0;
```

Define the standard gravitational parameter for Earth in m^3/s^2.

```
mu = 398600.4418e9;
```

The spacecraft stays on the initial orbit for one full period before initiating the transfer. Calculate the time corresponding to the first transfer point, `t1`, in seconds.

```
t1 = 2*pi*sqrt((a1^3)/mu);
```

The spacecraft must transfer to a coplanar circular orbit with an orbital radius of 10,000 km. Accordingly, set the semimajor axis, `a2`, of the final orbit in meters.

```
a2 = 10000e3;
```

Calculate the semimajor axis of the transfer orbit in meters.

```
aTransfer = (a1 + a2)/2;
```

The second transfer point occurs exactly one-half transfer orbital period after the first transfer point. Accordingly, calculate the transfer duration, `tTransfer`, in seconds.

```
tTransfer = pi*sqrt((aTransfer^3)/mu);
```

Calculate the time corresponding to the second transfer point, `t2`, in seconds.

```
t2 = t1 + tTransfer;
```

The mission ends after one orbital period on the final orbit. Accordingly, define the mission end time, `t2`, in seconds.

```
tf = t2 + (2*pi*sqrt((a2^3)/mu));
```

Calculate Maneuvers

The maneuvers at the first and second transfer points are defined by the maneuver times, delta-v magnitude, and delta-v direction. The maneuver times (`t1` and `t2`) are calculated in the previous topic. The remaining parameters are calculated in this topic.

Calculate Delta-V Magnitudes

Calculate the circular velocity on the initial orbit.

```
v1 = sqrt(mu/a1);
```

Calculate the velocity on the transfer orbit at the first transfer point.

```
vTransfer1 = sqrt(2*mu*((1/a1) - (1/(2*aTransfer))));
```

Calculate the delta-v magnitude at the first transfer point.

```
deltav1 = vTransfer1 - v1;
```

Calculate the circular velocity on the final orbit.

```
v2 = sqrt(mu/a2);
```

Calculate the velocity on the transfer orbit at the second transfer point.

```
vTransfer2 = sqrt(2*mu*((1/a2) - (1/(2*aTransfer))));
```

Calculate the delta-v magnitude at the second transfer point.

```
deltav2 = v2 - vTransfer2;
```

Calculate Delta-V Directions

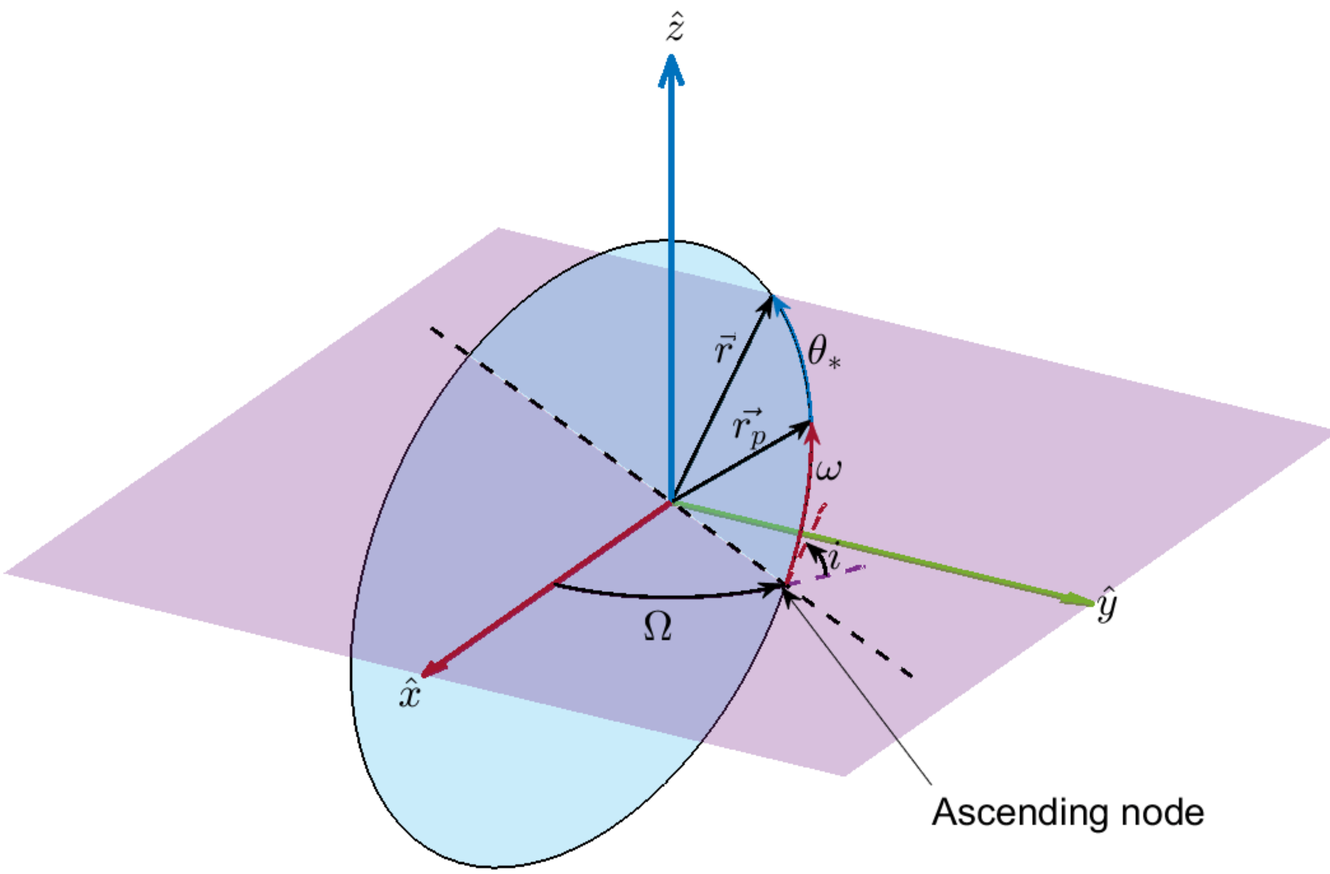
For circular orbits, the velocity direction in three-dimensional space is given by:

$$\hat{\mathbf{v}} = (-\cos\Omega\sin(\omega+\theta_*) - \sin\Omega\cos i\cos(\omega+\theta_*))\hat{x} + (-\sin\Omega\sin(\omega+\theta_*) + (\cos\Omega\cos i\cos(\omega+\theta_*)))\hat{y} + \sin i\cos(\omega+\theta_*)\hat{z}$$

where:

- $\hat{\mathbf{v}}$ is the velocity direction.
- Ω is the right ascension of ascending node.
- i is the inclination.
- ω is the argument of periapsis.
- θ_* is the true anomaly.

These quantities are defined in the International Celestial Reference Frame (ICRF) spanned by $\hat{x}$, $\hat{y}$, and $\hat{z}$ axes, as illustrated below.



Notes:

- The velocity vector is always tangential to the orbit. Because the transfer orbit is tangential to the initial and final circular orbits at the transfer points, the velocity directions on the circular and transfer orbits at these points are the same.
- The first transfer point is at the periapsis of the transfer orbit. The second transfer point is at the apoapsis of the transfer orbit. Therefore, the true anomalies on the transfer orbit corresponding to these transfer points are 0 degrees and 180 degrees respectively.
- Because the initial and final orbits are coplanar, the transfer orbit is also coplanar with these orbits. Therefore, the right ascension of ascending node and inclination are the same as those of the initial and final orbits.
- Because the true anomaly on the initial orbit corresponding to the first transfer point is the same as that on the transfer orbit (0 degrees), the argument of periapsis of the transfer orbit is the same as that of the initial orbit.

Define the true anomaly at the first transfer point on the transfer orbit in degrees.

```
thetaStar = 0;
```

Calculate the delta-v direction at the first transfer point.

```
deltav1Direction = [- cosd(raan1)*sind(w1 + thetaStar) - sind(raan1)*cosd(i1)*cosd(w1 + thetaSta
    - sind(raan1)*sind(w1 + thetaStar) + cosd(raan1)*cosd(i1)*cosd(w1 + thetaStar); ...
    sind(i1)*cosd(w1 + thetaStar)];
```

Define the true anomaly at the second transfer point on the transfer orbit in degrees.

```
thetaStar = 180;
```

Calculate the delta-v direction at the second transfer point.

```
deltav2Direction = [- cosd(raan1)*sind(w1 + thetaStar) - sind(raan1)*cosd(i1)*cosd(w1 + thetaStar); ...
    - sind(raan1)*sind(w1 + thetaStar) + cosd(raan1)*cosd(i1)*cosd(w1 + thetaStar); ...
    sind(i1)*cosd(w1 + thetaStar)];
```

Calculate Burn Durations

The preceding equations assume impulsive maneuvers, wherein the delta-v is achieved instantaneously. To approximate an impulsive delta-v in Simulink, assume a large thrust acceleration. The thrust for a given propellant mass flow rate $\dot{m}$ and specific impulse I_{sp} is given by:

$$T = \dot{m} g_0 I_{sp}$$

where g_0 is the acceleration due to gravity of the Earth equator. The quantity $g_0 I_{sp}$ is the exhaust velocity, v_e. Then, to approximate the impulsive delta-v, assume a propellant mass flow rate of 500 kg/s, a specific impulse of 400 s, and an initial spacecraft mass `m0` of 1000 kg. These assumptions create a thrust acceleration of 1962 m/s^2 (or 200 gs) when the propellant is full. This acceleration increases as the propellant gets depleted. The high thrust acceleration reduces the time required to achieve the calculated delta-v.

```
g0 = 9.81;
Isp = 400;
ve = g0*Isp;
mDot = 500;
m0 = 1000;
```

Based on the above thrust parameters, calculate the burn durations for each delta-v maneuver using the rocket equation. The rocket equation shows how the spacecraft mass (hence, the propellant mass) changes for a given delta-v. Suppose the initial spacecraft mass is m_0 and the mass after applying a delta-v of Δv reduces the mass to m_1. m_0 and m_1 are related by this rocket equation:

$$\frac{m_0}{m_1} = e^{\left(\frac{\Delta v}{v_e}\right)}$$

Calculate the mass `m1` after the maneuver at the first transfer point.

```
m1 = m0/exp(deltav1/ve);
```

Calculate the burn duration in seconds for the maneuver at the first transfer point.

```
burnDuration1 = (m0-m1)/mDot;
```

Calculate the mass `m2` after the maneuver at the second transfer point.

```
m2 = m1/exp(deltav2/ve);
```

Calculate the burn duration in seconds for the maneuver at the second transfer point.

```
burnDuration2 = (m1-m2)/mDot;
```

The burn durations are about 0.3 s and 0.24 s for the first and second transfer point, respectively. Next, examine the Simulink model that simulates the Hohmann transfer using the calculated maneuver parameters.

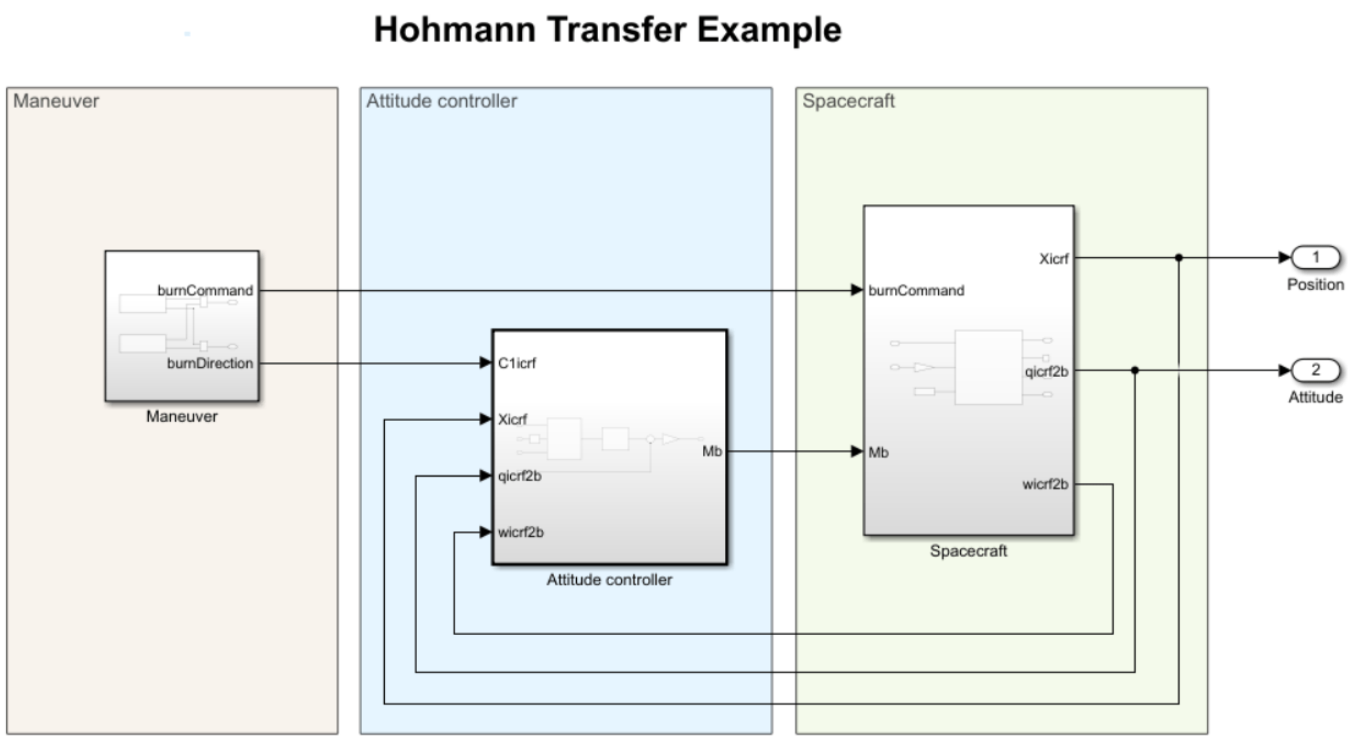
Examine Model

Examine the model that simulates the Hohmann transfer in Simulink by implementing the calculated maneuver parameters.

Open the Example Model

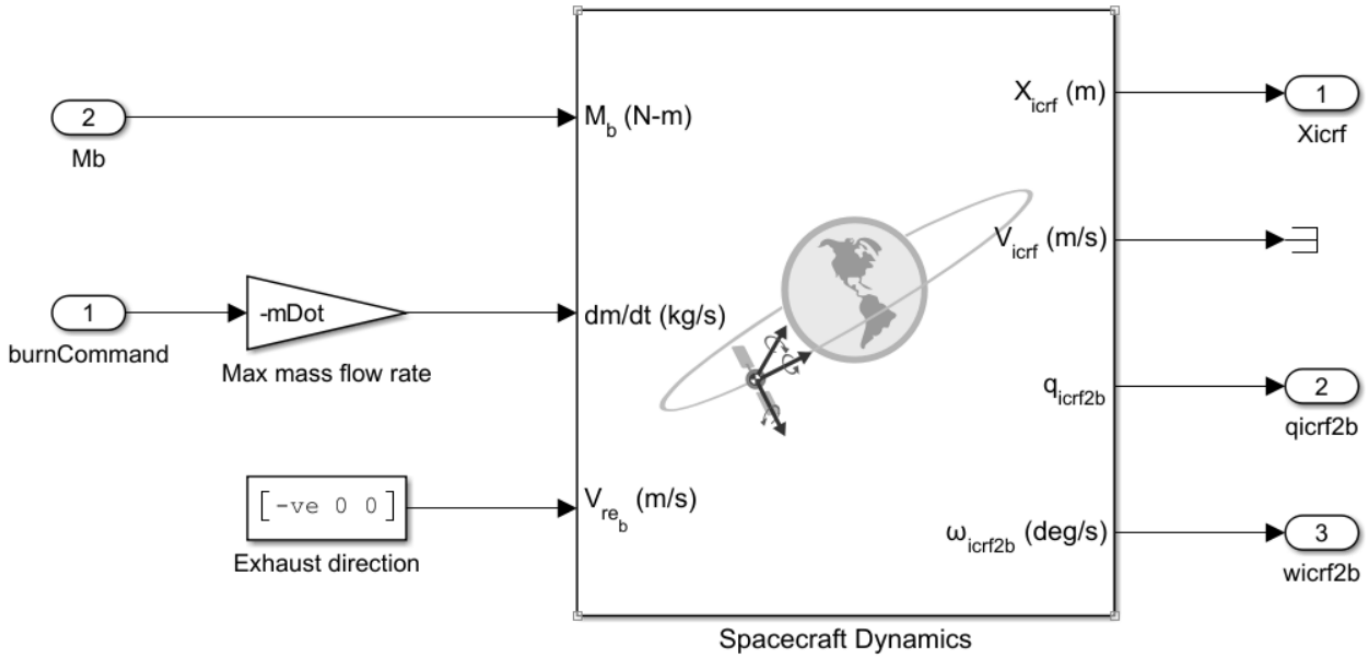
Open the model. At the top level, the model consists of the Maneuver, Attitude controller, and Spacecraft subsystems.

```
model = 'hohmannTransfer';
open_system(model);
```



Spacecraft Subsystem

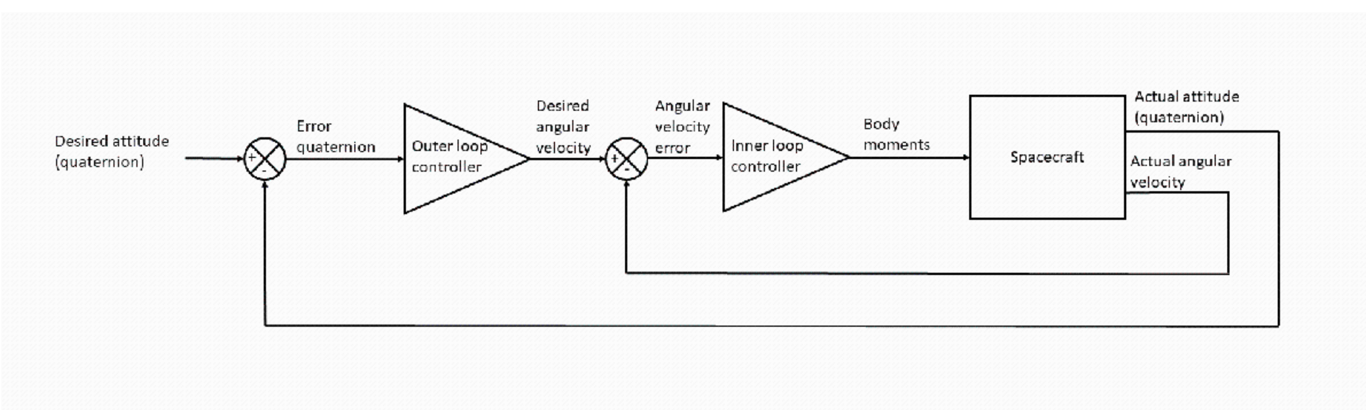
The Spacecraft subsystem models the spacecraft motion using the Spacecraft Dynamics block. The block accepts the external body moments, mass flow rate, and exhaust velocity as inputs. It outputs the ICRF position, attitude with respect to ICRF, and angular velocity in the body frame. The ICRF velocity output is unused.



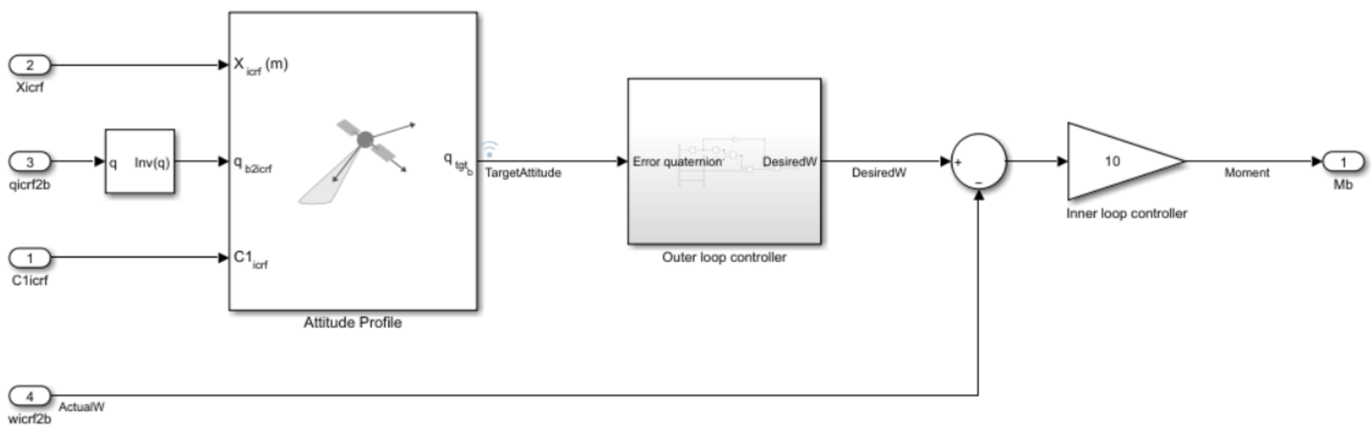
Attitude Controller Subsystem

The attitude controller subsystem calculates the body moments to align the x-axis of the spacecraft body (the primary alignment vector) with the burn direction (the primary constraint vector) for each maneuver and the z-axis of the spacecraft (the secondary alignment vector) with the z-axis of ICRF. The alignment of the primary vectors must be exact. This requirement implies that the exact alignment of the secondary vectors is not always possible. The goal in this instance is to minimize the misalignment of the secondary vectors.

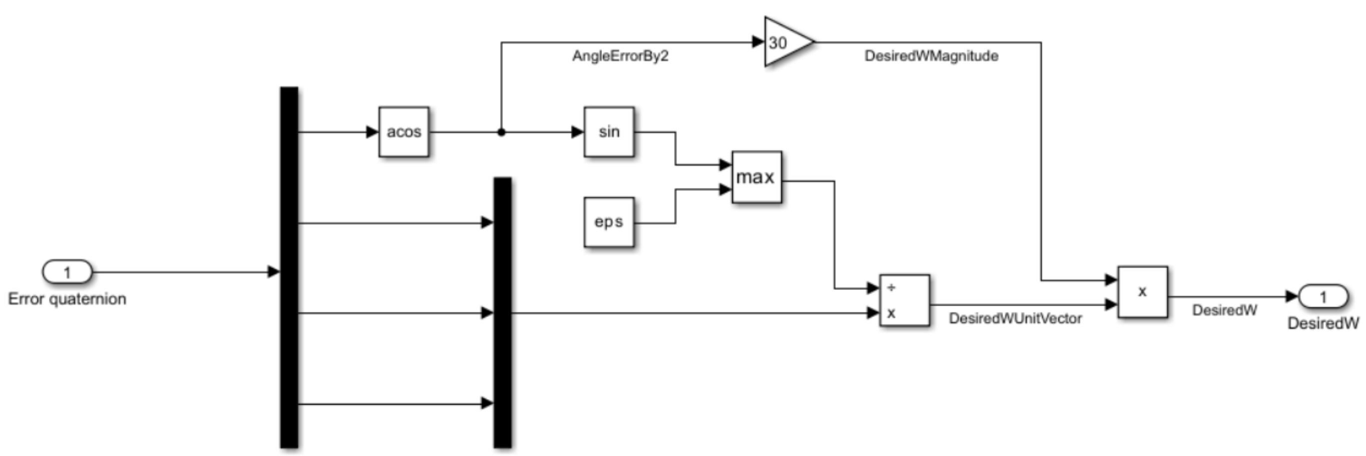
The high-level architecture of the controller is given here. It is a cascade controller, where the inner loop controls the angular velocity, and the outer loop controls the attitude.



The Attitude controller subsystem implementing the above architecture is shown here.



The Outer loop controller subsystem is shown here.

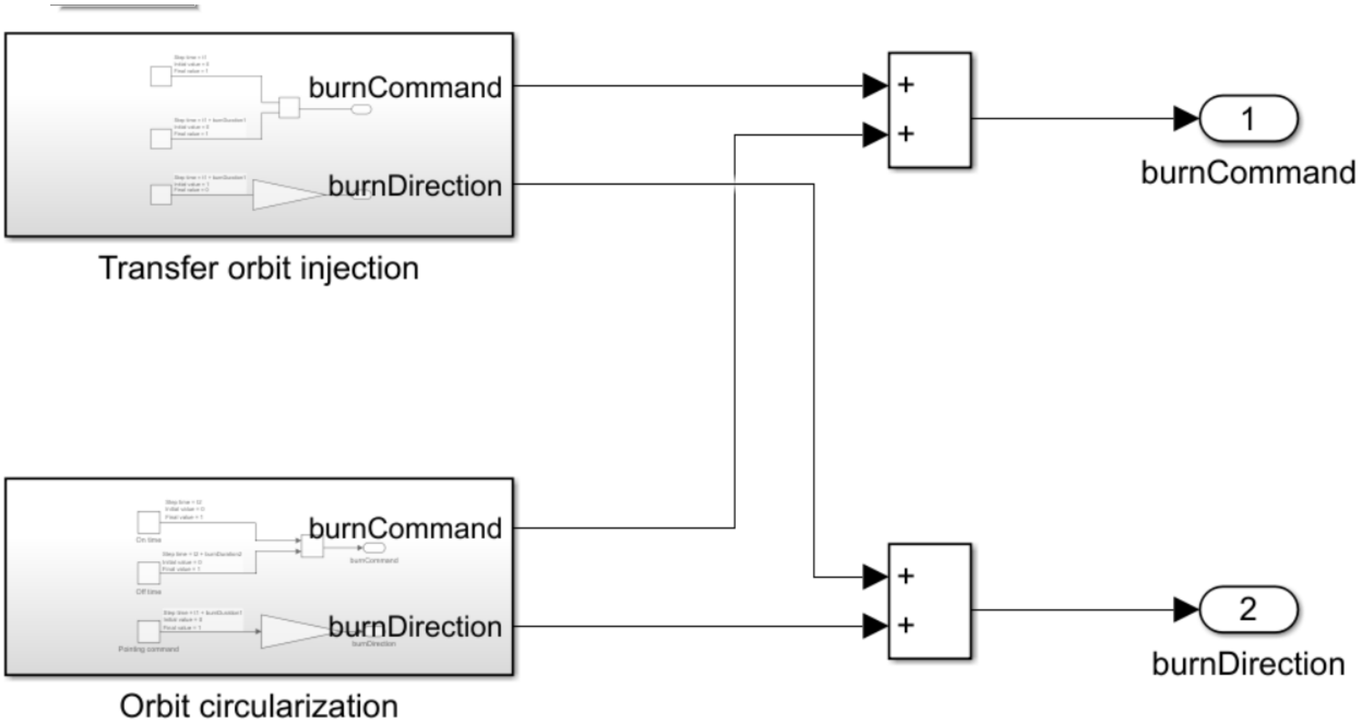


This controller extracts the rotation axis and angle from the error quaternion. The goal is to drive this angle to 0. A proportional controller calculates the angular velocity magnitude required to drive this angle to 0. The rotation axis calculated from the error quaternion is the desired unit angular velocity vector. This unit vector, combined with the desired angular velocity magnitude, constitutes the desired angular velocity vector, which in turn drives the inner loop.

The inner loop compares the desired angular velocity vector output from the outer loop controller against the actual angular velocity vector. The angular velocity error is fed to the inner loop controller, which is also implemented as a proportional controller. The output of this controller is the desired body moment vector, which is input to the Spacecraft Dynamics block in the Spacecraft subsystem.

Maneuver Subsystem

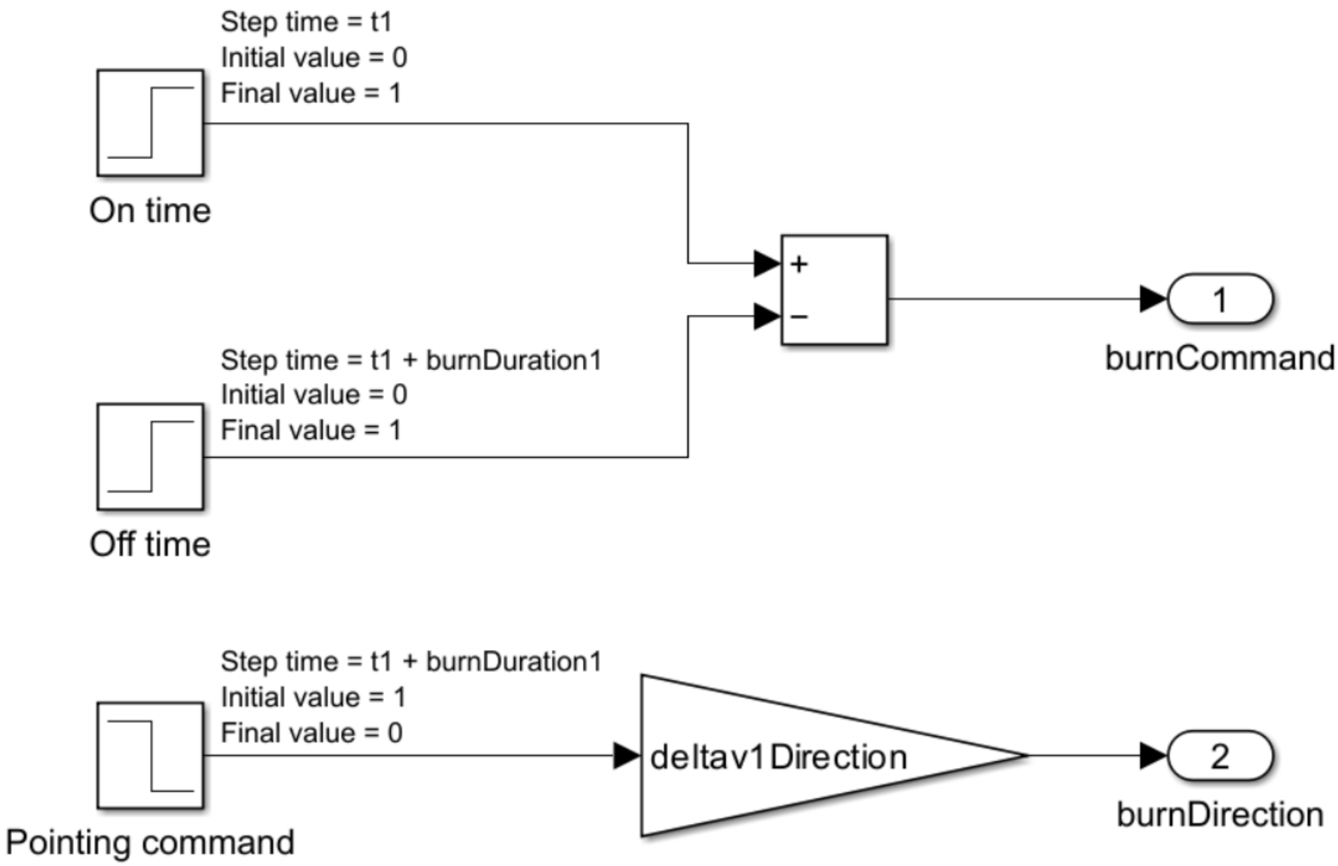
The Maneuver subsystem generates the burn commands (burnCommand) and burn direction (burnDirection) for the maneuvers at the transfer points. The Transfer orbit injection subsystem generates the burn command and direction for the first transfer point. The Orbit circularization subsystem generates the same for the second transfer point.



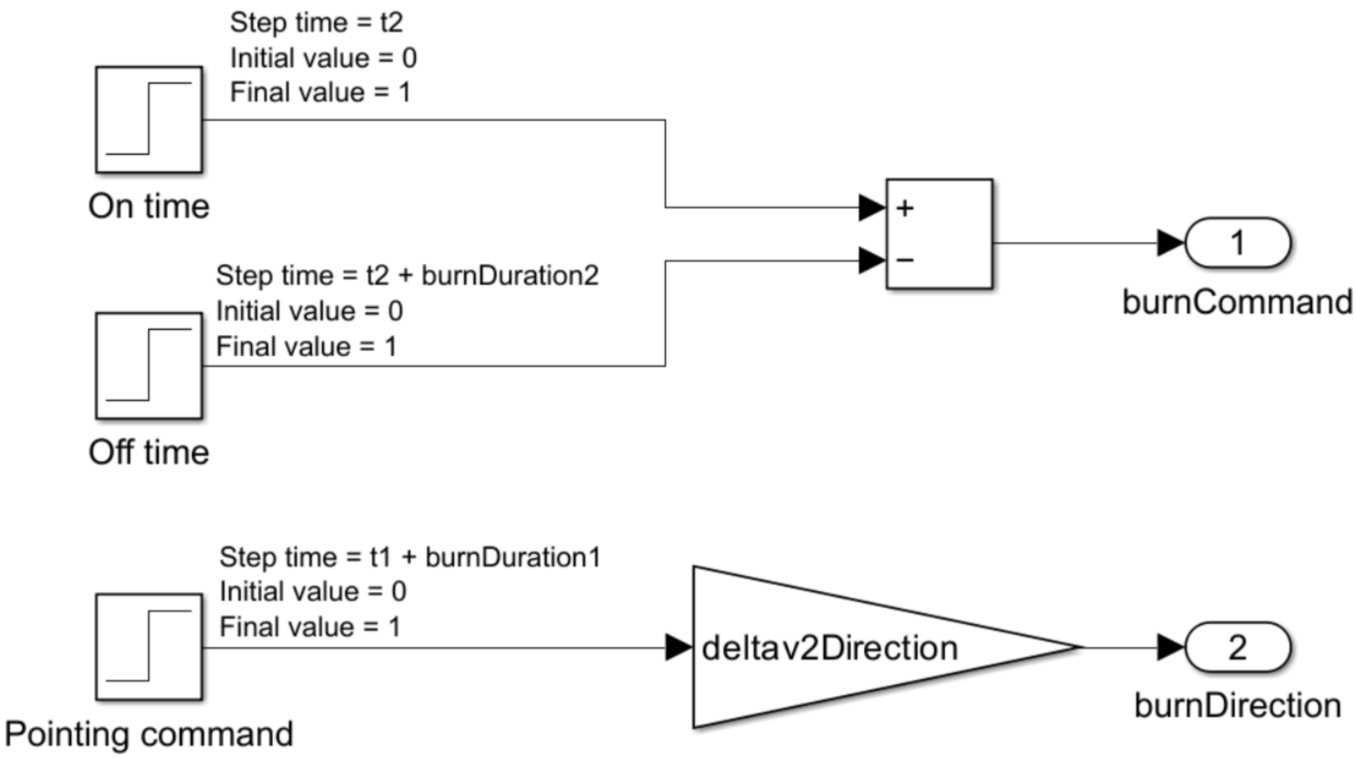
The burn commands take values 0 or 1. When the value is 1, the spacecraft propulsion system generates thrust. The burnCommand output drives the dm/dt input of the Spacecraft Dynamics block through the Max mass flow rate block in the Spacecraft subsystem. The `burnCommand` must equal 1 between `t1` and `t1 + burnDuration1` corresponding to the maneuver at the first transfer point and between `t2` and `t2 + burnDuration2` corresponding to the maneuver at the second transfer point. At all other times, it must equal 0.

The burn direction must initially equal the delta-v direction at the first transfer point (`burnDirection1`). After the first maneuver is complete (at `t1 + burnDuration1`), the burn direction must equal the delta-v direction at the second transfer point (`burnDirection2`) and maintain this direction for the duration of the simulation. The burnDirection output drives the Attitude controller subsystem.

The switching logic for the `burnCommand` and `burnDirection` implemented by the Transfer orbit injection subsystem is illustrated here.



The switching logic for the `burnCommand` and `burnDirection` implemented by the Orbit circularization subsystem is illustrated here.



Run Simulation

Run the simulation with the `sim` function. The model is configured to run in Rapid Accelerator mode for performance.

```
simOut = sim(model);
```

```
### Building the rapid accelerator target for model: hohmannTransfer
### Successfully built the rapid accelerator target for model: hohmannTransfer

Build Summary

Top model rapid accelerator targets built:

Model            Action                        Rebuild Reason
=====================================================================================================
hohmannTransfer  Code generated and compiled.  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 4.0375s
```

Extract Position and Attitude

Extract the position and attitude timeseries from the simulation results and convert them to timetables. These timetables will be imported into a satellite scenario object for visualizing the mission.

```
positionTT = timeseries2timetable(simOut.yout{1}.Values);
attitudeTT = timeseries2timetable(simOut.yout{2}.Values);
```

Visualize Mission

Import the simulation results into a satellite scenario to visualize the mission on a satellite scenario viewer.

Create a Satellite Scenario Object

Create a satellite scenario object based on the initial time and simulation duration. Set the sample time to 60 seconds.

```
startTime = initialTime;
stopTime = startTime + seconds(tf);
sampleTime = 60;
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

Add a Satellite to the Scenario

Add a satellite to the scenario using the position timetable.

```
sat = satellite(sc,positionTT,Name = "Spacecraft");
```

Define Attitude Profile Based on Simulation

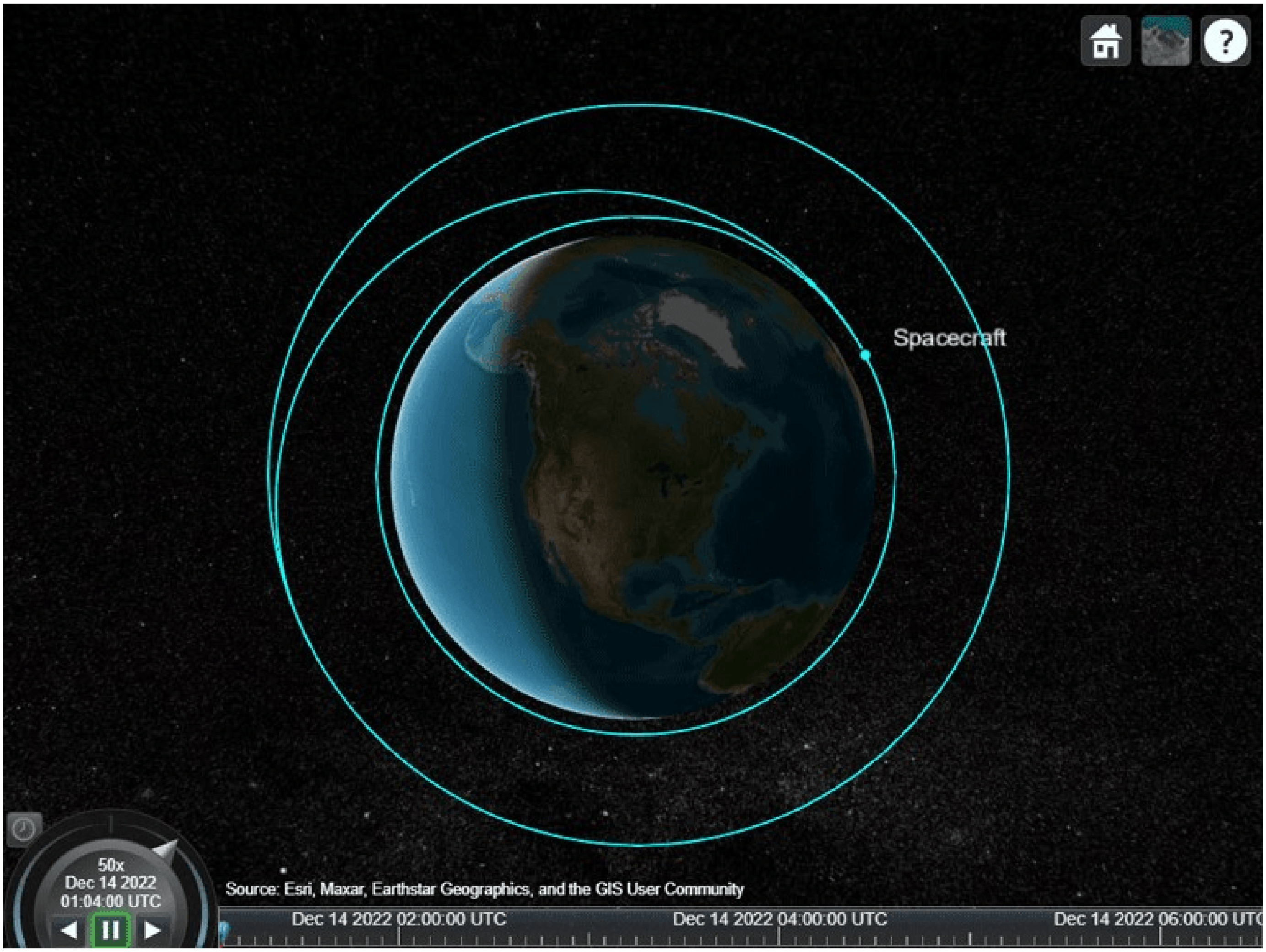
Use `pointAt` to define the attitude profile of the satellite in the scenario based on the attitude timetable.

```
pointAt(sat,attitudeTT);
```

Launch Satellite Scenario Viewer

Launch a satellite scenario viewer to visualize the scenario. Set the camera reference frame to **Inertial** to inertially fix the camera orientation. Left-click and hold anywhere inside the satellite scenario viewer window to pan the camera. Adjust the zoom level using the scroll wheel.

```
% Launch a satellite scenario viewer
v = satelliteScenarioViewer(sc,CameraReferenceFrame = "Inertial");
```

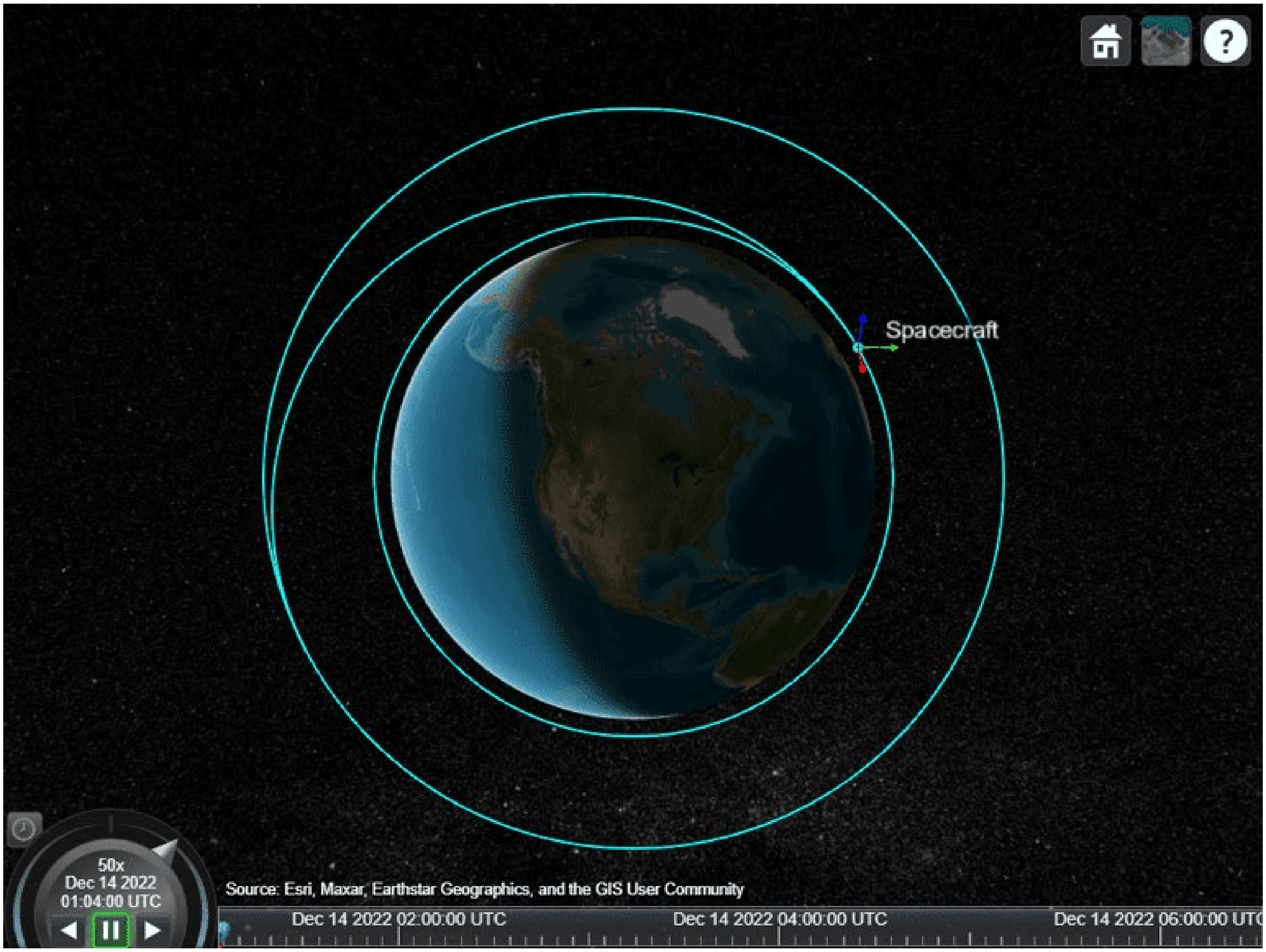


The viewer shows the trajectory flown by the spacecraft. The initial orbit, the transfer orbit, and the final orbit are clearly visible.

Visualize Body Coordinate Axes

Use `coordinateAxes` to visualize the body coordinate axes of the spacecraft. This enables you to visualize the attitude of the spacecraft. The red, green, and blue arrows are the body x-, y-, and z-axes respectively.

```
coordinateAxes(sat);
```



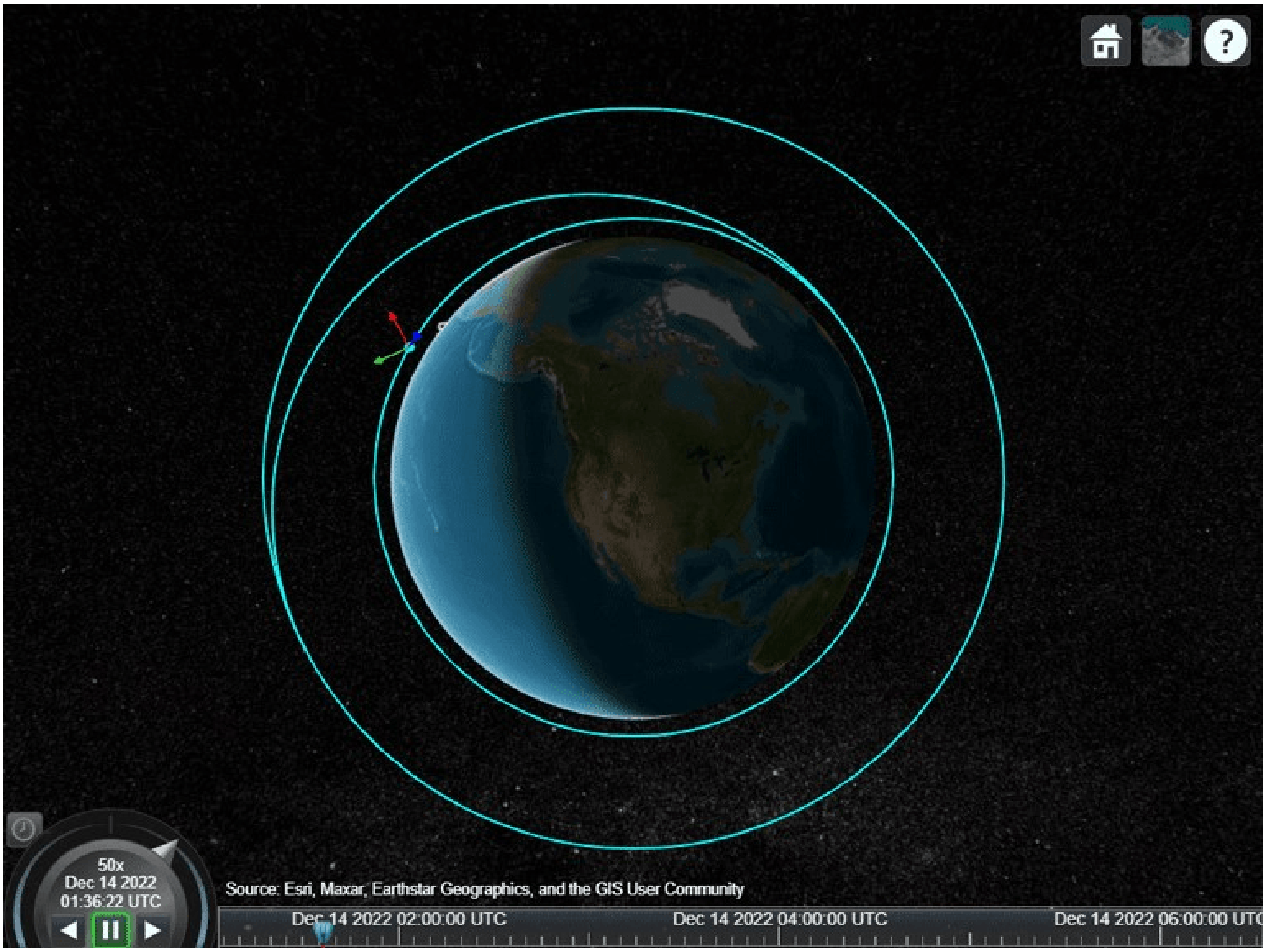
Play Scenario

Animate the scenario using `play`.

```
play(sc);
```

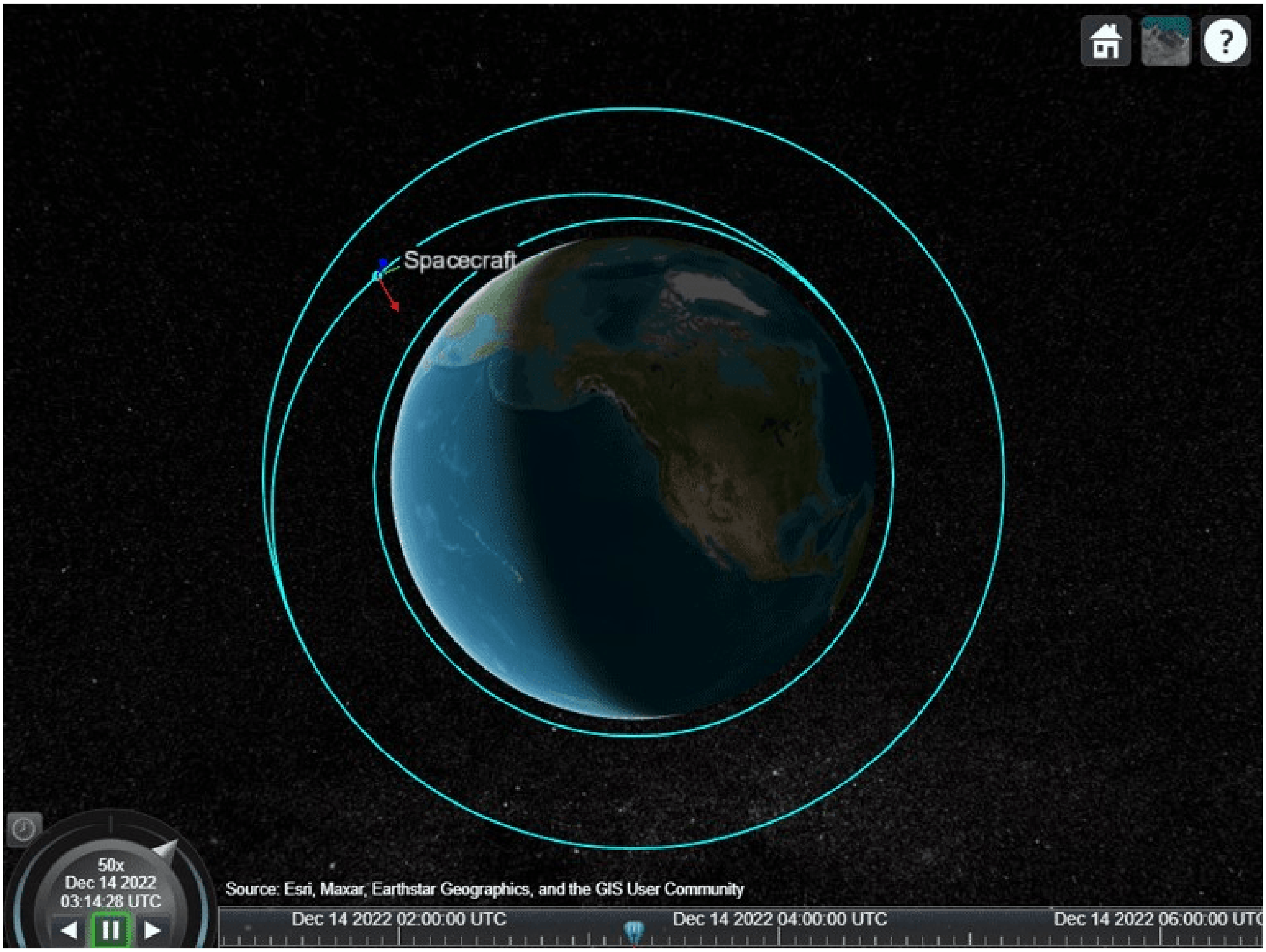
Note how the spacecraft attitude changes in preparation for the first maneuver that occurs after one orbital period.

```
v.CurrentTime = sc.StartTime + seconds(t1/3);
```



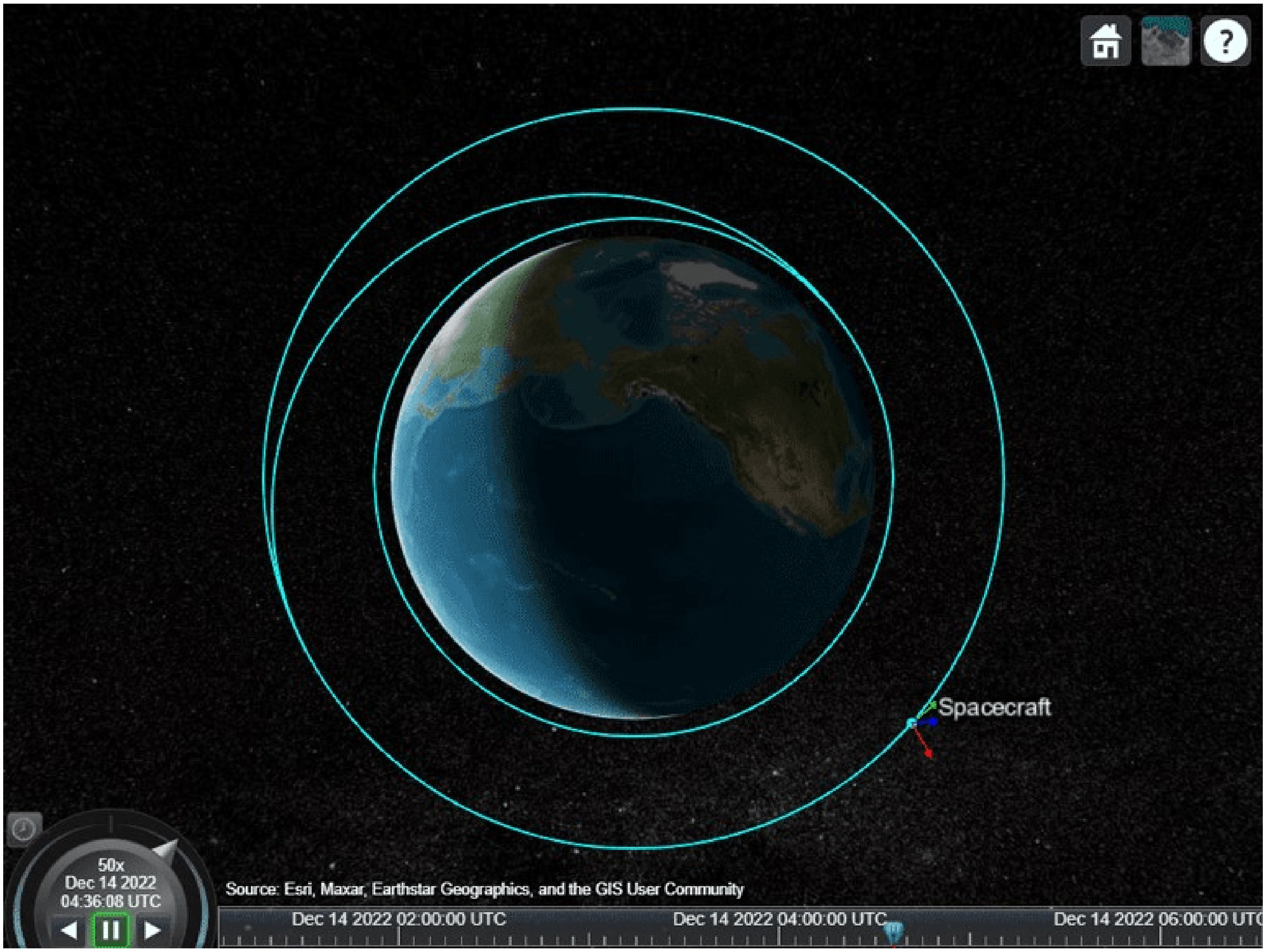
After the first maneuver, the spacecraft reorients in preparation for the second maneuver.

```
v.CurrentTime = sc.StartTime + seconds(t1 + 2000);
```



After the second maneuver, the spacecraft is on the final orbit. It continues to hold the same attitude used during the second maneuver.

```
v.CurrentTime = sc.StartTime + seconds(t2 + 3000);
```



Conclusion and Next Steps

This example demonstrates how to model Hohmann transfer using the Spacecraft Dynamics block and visualize the trajectory using satellite scenario. Of note:

- The initial and final orbits are assumed to be circular and coplanar, which result in a transfer orbit that is also coplanar.
- The spacecraft attitude is controlled for each maneuver in the transfer using an attitude controller. This controller uses the Attitude Profile block to calculate the error between the desired attitude and the actual attitude.
- The maneuvers are calculated assuming impulsive delta-v's, where the desired velocity change is assumed to be achieved instantaneously.
- The Simulink model approximates the impulsive delta-v by allowing the spacecraft to generate a large amount of thrust, reducing the burn durations.
- The thrust is modeled using the specific impulse and propellant mass flow rate.

To improve the example:

- Model finite burns by using a more realistic propellant mass flow rate. A finite burn results in a final orbit that is slightly different than the desired orbit unless the maneuver calculations are updated to accommodate such burns. Such calculations incorporate optimal control theories and involve numerical methods as opposed to analytical methods.
- Model reaction thrusters and use the calculated control moments to determine the control allocation among the thrusters.
- Assume elliptical initial and final orbits. Note that the transfer orbit must still be tangential to the two orbits. However, the true anomalies on the transfer orbit at the two transfer points might not be 0 and 180 degrees.

See Also

Blocks
Spacecraft Dynamics | Attitude Profile

Objects
`satelliteScenario`

Space Rendezvous and Docking

This example shows the workflow required to plan maneuvers to enable a spacecraft in orbit to rendezvous and dock with another spacecraft. The rendezvous operation consists of a set of orbital maneuvers during which two spacecraft arrive at the same orbit and approach each other from close proximity. The *target* spacecraft is passive. The *chaser* spacecraft performs the maneuvers to rendezvous with the target. The docking maneuver consists of the chaser maneuvering further closer to the target and docking with it. In this example, the rendezvous maneuvers are planned using Optimization Toolbox™. The docking maneuver is performed by a closed loop controller that commands the force that needs to be provided by the translational reaction thrusters along each body axis of the chaser. The mission is simulated in Simulink® software and visualized in a satellite scenario. This example uses:

- Aerospace Blockset™ Spacecraft Dynamic**s** block
- Aerospace Blockset Attitude Profile block
- Aerospace Toolbox™ `satelliteScenario` object
- Optimization Toolbox™ `fsolve` function
- Optimization Toolbox™ `fmincon` function
- Stateflow™ Chart block

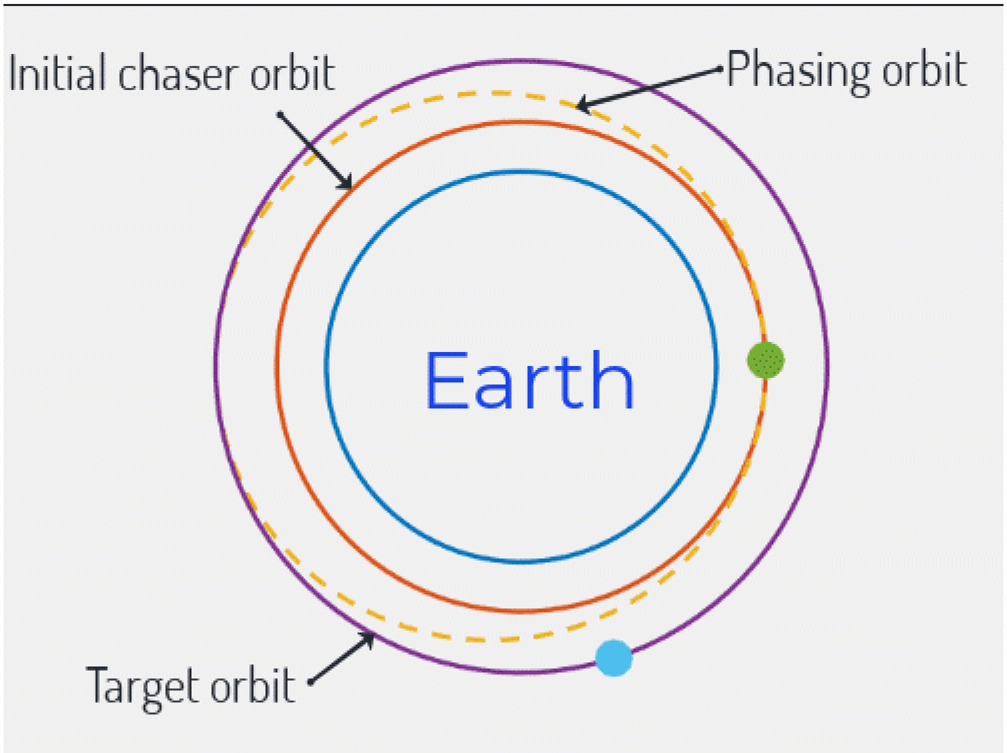
Mission Overview

The mission consists of two phases, rendezvous and docking.

Rendezvous Phase

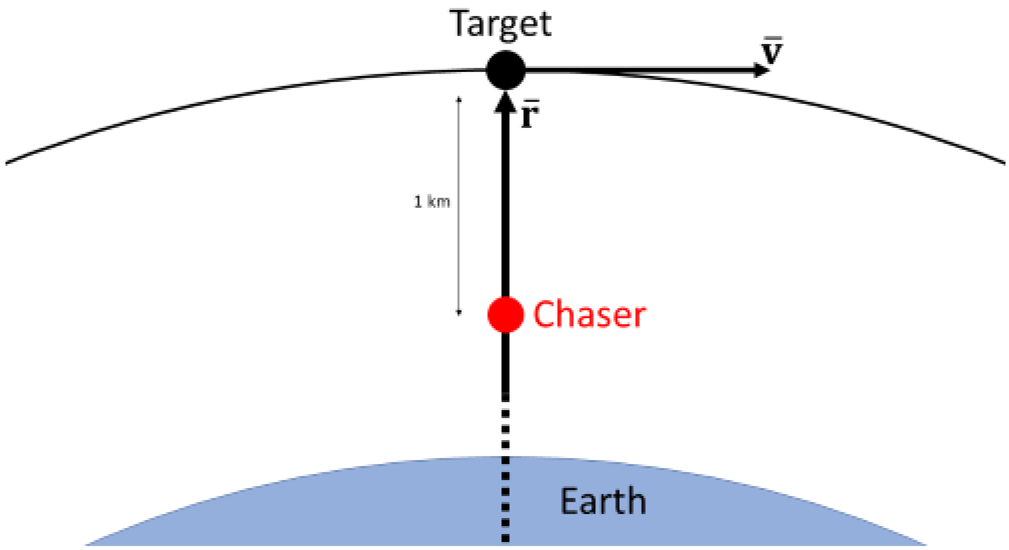
The rendezvous phase consists of a series of maneuvers that enables the chaser to approach the target to a very close distance. Before rendezvous can commence, the initial orbit of the chaser and target must be on the same plane. This alignment is typically accomplished by timing the launch of the chaser so that the chaser is inserted into the correct orbital plane. Otherwise, plane change maneuvers must be performed at the intersection of the chaser and target orbital planes, which typically incurs a substantial delta-v cost. Assuming the orbital planes are aligned, rendezvous can be performed by placing the chaser on a phasing orbit. The phasing orbit approaches close to the target orbit (in the order of a few kilometers) and has a slightly different orbital period. Because of the difference in the orbital periods, the chaser and the target approach closer to one another after each orbit. If the insertion into the phasing orbit is timed correctly, after several orbits, both chaser and target will arrive at the closest point in their respective orbits at the same time. Upon arrival at this point, rendezvous has been accomplished.

This animation shows the rendezvous sequence.



- The chaser is initially on a lower orbit compared to the target.
- The first maneuver places the chaser on the phasing orbit. After several orbits, the chaser and the target approach close to one another.
- When the chaser and the target are at their closest, the chaser performs a second maneuver to cancel out the relative velocity.

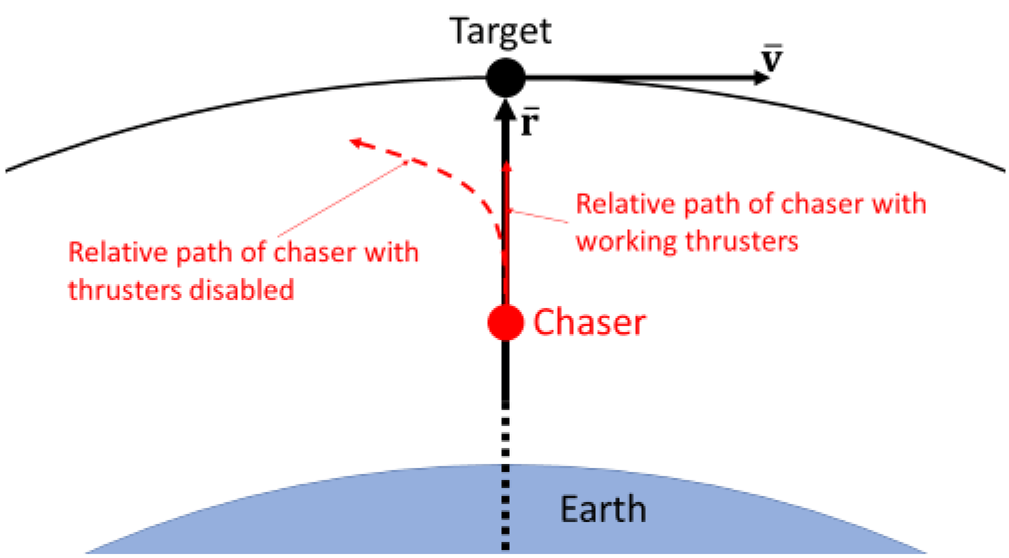
The rendezvous conditions require the chaser to be placed 1 km below the target along the radial position vector, $\bar{\mathbf{r}}$, of the target. In addition, the relative velocity between the two spacecraft must be 0 m/s. From this point, the docking operations commence. In the diagram, $\bar{\mathbf{v}}$ is the ICRF velocity of the target. At the instant rendezvous is accomplished, the velocity of the chaser must also equal $\bar{\mathbf{v}}$.



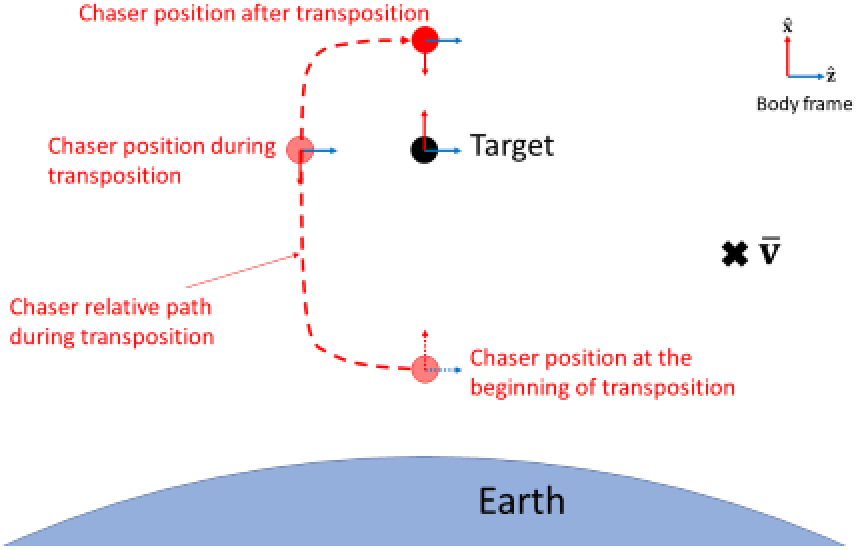
Docking Phase

After rendezvous is accomplished, docking operations are commenced in three phases:

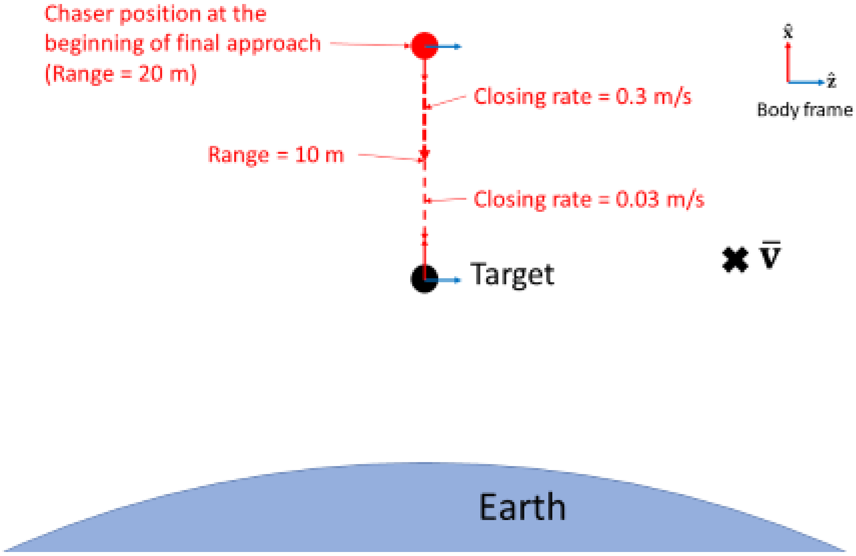
- **Initial approach**: The initial approach or R-bar approach, requires the chaser to move toward the target along the radial position vector, or $\bar{\mathbf{r}}$, of the target using translational thrusters at about 0.3 m/s. During this phase, the relative distance is brought down to 100 m. The R-bar approach requires the relative velocity component of the chaser with respect to the target that is perpendicular to $\bar{\mathbf{r}}$ to be maintained at 0. As the chaser approaches the target from below, its orbital angular velocity drops. The chaser compensates by firing the translational thrusters perpendicular to $\bar{\mathbf{r}}$. If the thrusters fail, the chaser starts lagging behind the target. This lag is an inherent safety feature of the R-bar approach because it eliminates the possibility of a collision with the target if the chaser thrusters fail.



- **Transposition**: The chaser reorients to docking attitude and maneuvers around the target to position itself about 20 m in front of the docking port of the target. If the attitude of the target is such that a direct path between the initial and final position during the transposition phase causes a collision, the chaser maneuvers around the target. The maneuver is done such that the target is offset by 20 m along the body *z* axis of the chaser. The offset about the body *y* axis of the chaser is always commanded to be 0 m. As a result, the transposition maneuver happens on the *x*-*z* plane of the body frame of the chaser while it is in its docking attitude.



- **Final approach**: The chaser closes in on the target at 0.3 m/s. When the range to the target reaches 10 m, the chaser reduces its closing rate to 0.03 m/s until their respective docking ports meet each other.



Mission Setup

The mission begins on August 8, 2022, 12:00 AM UTC.

```
epochYear = 2022;
epochMonth = 8;
epochDay = 6;
epochHour = 0;
epochMinute = 0;
epochSecond = 0;
epoch = datetime(epochYear,epochMonth,epochDay, ...
    epochHour,epochMinute,epochSecond);
```

The initial orbital elements of the chaser are:

- Semimajor axis (a_1) = 7,500 km
- Eccentricity = 0.001
- Inclination = 30.1 degrees
- Right ascension of ascending node = 60.1 degrees
- Argument of periapsis = 120 degrees
- True anomaly = 30 degrees

```
initialChaserSemimajorAxis = 7500000;              % m
initialChaserEccentricity = 0.001;
initialChaserInclination = 30.1;                   % deg
initialChaserRightAscensionofAscendingNode = 60.1; % deg
initialChaserArgumentOfPeriapsis = 120;            % deg
initialChaserTrueAnomaly = 30;                     % deg
initialChaserOscullatingElements = [initialChaserSemimajorAxis, ...
    initialChaserEccentricity, ...
    initialChaserInclination, ...
    initialChaserRightAscensionofAscendingNode, ...
    initialChaserArgumentOfPeriapsis, ...
    initialChaserTrueAnomaly];
```

The initial orbital elements of the target are:

- Semimajor axis (a_1) = 8,000 km
- Eccentricity = 0.0005
- Inclination = 30 degrees
- Right ascension of ascending node = 60 degrees
- Argument of periapsis = 120 degrees
- True anomaly = 310 degrees

```
initialTargetSemimajorAxis = 8000000;            % m
initialTargetEccentricity = 0.0005;
initialTargetInclination = 30;                   % deg
initialTargetRightAscensionofAscendingNode = 60; % deg
initialTargetArgumentOfPeriapsis = 120;          % deg
initialTargetTrueAnomaly = 310;                  % deg
initialTargetOscullatingElements = [initialTargetSemimajorAxis, ...
    initialTargetEccentricity, ...
    initialTargetInclination, ...
```

```
    initialTargetRightAscensionofAscendingNode, ...
    initialTargetArgumentOfPeriapsis, ...
    initialTargetTrueAnomaly];
```

Both spacecraft control their attitude by aligning their *x*-axes towards the commanded direction, which is the primary constraint vector. Therefore, the *x*-axis of each spacecraft constitutes the primary alignment vector.

```
primaryAlignmentVector = [1;0;0];
```

The spacecraft also attempt to minimize the alignment of their *z* axes towards a secondary direction, which is the secondary constraint vector. Therefore, the *z* axis of each spacecraft constitutes the secondary alignment vector.

```
secondaryAlignmentVector = [0;0;1];
```

Throughout the mission, the target spacecraft body axis remains aligned with the International Celestial Reference Frame (ICRF). Therefore, the primary and secondary constraint vectors for the target spacecraft are the ICRF *x*- and *z*- axes respectively.

```
primaryConstraintVectorTarget = [1;0;0];
secondaryConstraintVectorTarget= [0;0;1];
```

Accordingly, calculate the scalar-first quaternion of the target spacecraft using the `SpaceRendezvousAndDockingExampleCalculateAttitude` helper function.

```
targetAttitude = SpaceRendezvousAndDockingExampleCalculateAttitude( ...
    primaryAlignmentVector, ...
    primaryConstraintVectorTarget, ...
    secondaryAlignmentVector, ...
    secondaryConstraintVectorTarget);
```

The primary and secondary constraint vectors for the chaser are dependent on the mission phase.

Mission Phase	Primary Constraint Vector	Secondary Constraint Vector
Rendezvous	Delta-v direction for the first maneuver until this maneuver is complete. After this, it is the delta-v direction for the second maneuver.	ICRF z-axis
Docking (initial approach)	Radial position vector of target	Orbital angul
Docking (transposition)	Negative x-axis of target body frame	z-axis of target body frame
Docking (final approach)	Negative x-axis of target body frame	z-axis of target body frame

The rendezvous condition requires the chaser to position itself 1 km below the target spacecraft along the target's radial position vector. The relative velocity between the two must be 0.

```
finalRelativeRadialDiscance = -1000; % m
finalRelativeVelocityICRF = [0;0;0]; % m/s
```

The chaser must achieve rendezvous conditions using two maneuvers. Between the two maneuvers, the chaser is in coast phase. After the second maneuver is complete, the rendezvous conditions must be satisfied. During the coast phase, the periapsis of the chaser must not dip below 6578.137 km, which translates to an altitude of roughly 200 km.

```
minimumChaserPeriapsisBetweenBurns = 6578137; % m
```

The maneuvers must meet these conditions:

• For each maneuver, the delta-v components must not exceed 300 m/s.

• The first maneuver must be performed no earlier than 200 s after mission start time to allow for the chaser attitude to stabilize for the first maneuver.

• The first maneuver must be performed no later than one day after the mission start time.

• The second maneuver must be performed no later than one day after the first maneuver.

```
lowerBoundForFirstBurnTime = 200;                     % s
lowerBoundForDeltaV = [-300;-300;-300];               % m/s
lowerBoundForSecondBurnTimeAfterFirstBurn = 0;        % s
upperBoundForFirstBurnTime = 24*3600;                 % s
upperBoundForDeltaV = [300;300;300];                  % m/s
upperBoundForSecondBurnTimeAfterFirstBurn = 24*3600; % s
```

After the rendezvous conditions are satisfied, the chaser enters docking phase. During the initial approach, the closing rate is 0.3 m/s.

```
initialApproachClosingRate = 0.3; % m/s
```

During transposition, if the direct path between initial and final position of chaser poses a risk of collision, the chaser maneuvers around the target so that the target is offset by 20 m along the body *z*-axis of the chaser. After transposition, the chaser must be positioned 20 m in front of the target.

```
collisionAvoidanceZOffset = 20;       % m
transpositionFinalPosition = [20;0;0]; % m
```

During final approach, the closing rate is 0.3 m/s when the range to target is greater than 10 m, and 0.03 m/s otherwise.

```
finalApproachClosingRate1 = 0.3;  % m/s
finalApproachClosingRate2 = 0.03; % m/s
```

Optimization of Maneuvers

You can calculate the maneuvers required to facilitate rendezvous are calculated using Optimization Toolbox. The rendezvous maneuvers involve firing the main propulsion system and are approximated as impulsive maneuvers. You can formulate the rendezvous problem as a nonlinear constrained optimization problem. The goal of the problem is to minimize the delta-v magnitude caused by the impulsive maneuvers while satisfying the constraints defined by:

- Initial orbital elements of the chaser on page 9-250 and the target on page 9-250
- Rendezvous conditions on page 9-251 specified by relative position and velocity between the chaser and the target
- Periapsis constraint on page 9-251 on the chaser between the maneuvers
- Lower and upper bounds on page 9-252 on delta-v components and times when the maneuvers are performed

The design variables of the optimization problem are the maneuver parameters consisting of:

- First burn time
- Delta-v for first burn along ICRF *x*-axis
- Delta-v for first burn along ICRF *y*-axis
- Delta-v for first burn along ICRF *z*-axis
- Second burn time counted from the first burn time
- Delta-v for second burn along ICRF *x*-axis
- Delta-v for second burn along ICRF *y*-axis
- Delta-v for second burn along ICRF *z*-axis

To aid in solution convergence, the position, velocity, and time are scaled to ensure that their magnitudes are roughly of the same order.

```
scaleTime = 50000;  % s
scalePos = 6378137; % m
scaleVel = 8000;    % m/s
```

Define the scaled lower and upper bounds on the maneuver parameters.

```
maneuverParameterLowerBound = ...
    [lowerBoundForFirstBurnTime/scaleTime; ...
    lowerBoundForDeltaV/scaleVel; ...
    lowerBoundForSecondBurnTimeAfterFirstBurn/scaleTime; ...
    lowerBoundForDeltaV/scaleVel];
maneuverParameterUpperBound = ...
    [upperBoundForFirstBurnTime/scaleTime; ...
    upperBoundForDeltaV/scaleVel; ...
    upperBoundForSecondBurnTimeAfterFirstBurn/scaleTime; ...
    upperBoundForDeltaV/scaleVel];
```

The sum of norm-squared of each delta-v constitutes the objective function and is defined in the file `SpaceRendezvousAndDockingExampleObjectiveFcn.m` . The nonlinear equality constraints defined by the rendezvous conditions and the nonlinear inequality constraint defined by the periapsis of the chaser between the rendezvous maneuvers are defined in the file `SpaceRendezvousAndDockingExampleConstraintFcn.m`.

This example performs the optimization using the `fmincon` function, which requires linear equality and inequality constraints to be defined. Because the defined optimization problem defined above does not contain such linear constraints, assign empty values to these.

```
linearInequalityConstraintA = [];
linearInequalityConstraintb = [];
linearEqualityConstraintA = [];
linearEqualityConstraintb = [];
```

You must supply `fmincon` with a guess for the optimal maneuver parameters. If this initial guess is arbitrary, fmincon may not converge to a solution. To mitigate this outcome, generate the initial guess using the `fsolve` function, in which no optimization is performed and the inequality constraints and bounds on the maneuver parameters are ignored. You can reformulate the optimization problem to just satisfy the equality constraints in the file `SpaceRendezvousAndDockingExampleConstraintFcn.m`. This reduces the optimization problem to a root solving problem. Note that fsolve also requires an initial guess. However, because the root solving problem has no inequality constraints or bounds on the maneuver parameters, you can solve it using `fsolve` with relative ease, even with an arbitrary initial guess. The solution of the root solving problem is usually a good initial guess for the original optimization problem because it already obeys the orbital mechanics and satisfies the equality constraints.

Define the arbitrary initial guess for `fsolve`.

```
firstBurnTime = 200;                 % s
firstDeltaV = [100;100;100];         % m/s
secondBurnTimeAfterFirstBurn = 80000; % s
secondDeltaV = [100;100;100];        % m/s
initialGuess = [firstBurnTime/scaleTime; ...
    firstDeltaV/scaleVel; ...
    secondBurnTimeAfterFirstBurn/scaleTime; ...
    secondDeltaV/scaleVel];
```

Define `fsolve` configuration using `optimoptions`.

```
options = optimoptions( ...
    'fsolve', ...
    MaxFunctionEvaluations = 1000000, ...
    MaxIterations = 10000, ...
    Algorithm = 'levenberg-marquardt', ...
    FunctionTolerance = 1e-6, ...
    StepTolerance = 1e-10);
```

Calculate the maneuver parameters using `fsolve`. The root solving problem often has multiple solutions. The actual solution that `fsolve` converges to depends on the initial guess.

```
needToReturnInequalityConstraint = false;
maneuver = fsolve( ...
    @SpaceRendezvousAndDockingExampleConstraintFcn, ...
    initialGuess, ...
    options, ...
    epoch, ...
    initialChaserOscullatingElements, ...
    initialTargetOscullatingElements, ...
    finalRelativeRadialDiscance, ...
    finalRelativeVelocityICRF, ...
    minimumChaserPeriapsisBetweenBurns, ...
    needToReturnInequalityConstraint, ...
    scalePos, ...
    scaleVel, ...
    scaleTime);
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
```

The initial guess for `fmincon` is the solution of `fsolve`.

```
initialGuess = maneuver;
```

Define `fmincon` configuration parameters using `optmimoptions`.

```
options = optimoptions( ...
    'fmincon', ...
```

```
    Algorithm = 'sqp',...
    MaxFunctionEvaluations = 1000000, ...
    MaxIterations = 10000, ...
    StepTolerance = eps, ...
    OptimalityTolerance = 1e-2);
```

Calculate the optimal maneuver parameters using `fmincon`. The optimization problem contains multiple local minima. `fmincon` converges to one of these minima depending on the initial guess provided by `fsolve`, which in turn depends on the arbitrary initial guess provided to `fsolve`.

```
needToReturnInequalityConstraint = true;
optimizedManeuver = fmincon( ...
    @SpaceRendezvousAndDockingExampleObjectiveFcn, ...
    initialGuess, ...
    linearInequalityConstraintA, ...
    linearInequalityConstraintb, ...
    linearEqualityConstraintA, ...
    linearEqualityConstraintb, ...
    maneuverParameterLowerBound, ...
    maneuverParameterUpperBound, ...
    @SpaceRendezvousAndDockingExampleConstraintFcn, ...
    options, ...
    epoch, ...
    initialChaserOscullatingElements, ...
    initialTargetOscullatingElements, ...
    finalRelativeRadialDiscance, ...
    finalRelativeVelocityICRF, ...
    minimumChaserPeriapsisBetweenBurns, ...
    needToReturnInequalityConstraint, ...
    scalePos, ...
    scaleVel, ...
    scaleTime);
```

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```

Rendezvous Maneuver Parameter Extraction

Extract the rendezvous maneuver parameters to calculate the burn direction and duration for each maneuver.

Extract the maneuver times for each rendezvous maneuver. These times constitute the respective burn start time for each maneuver.

```
burnTime1 = optimizedManeuver(1)*scaleTime;
burnTime2 = (optimizedManeuver(1) + optimizedManeuver(5))*scaleTime;
```

Extract the delta-v vectors for each maneuver.

```
deltaV1Vector = optimizedManeuver(2:4)*scaleVel;
deltaV2Vector = optimizedManeuver(6:8)*scaleVel;
```

Calculate the delta-v magnitudes for each maneuver.

```
deltaV1 = norm(deltaV1Vector);
deltaV2 = norm(deltaV2Vector);
```

Calculate the burn directions in ICRF for each rendezvous maneuver. These directions are simply the unit vector along each delta-v vector.

```
burnDirection1 = deltaV1Vector/norm(deltaV1Vector);
burnDirection2 = deltaV2Vector/norm(deltaV2Vector);
```

The rendezvous maneuvers calculated by `fmincon` assume impulsive maneuvers, in which the delta-v is achieved instantaneously. To approximate an impulsive maneuver in Simulink model, assume a large propellant mass flow rate for the main propulsion system.

```
mDot = 7500; % kg/s
```

Define the initial chaser mass and specific impulse of the main propulsion system. The values are required to calculate the burn durations for each rendezvous maneuver.

```
m0 = 1000; % kg
Isp = 400; % s
```

Calculate the main propulsion system exhaust velocity, which will be used to calculate chaser mass after each rendezvous maneuver burn.

```
ve = 9.81*Isp; % m/s
```

Calculate the chaser mass after the first maneuver burn.

```
m1 = m0/exp(deltaV1/ve);
```

Calculate the chaser mass after the second maneuver burn.

```
m2 = m1/exp(deltaV2/ve);
```

Using the change in mass after each burn and the mass flow rate, calculate the burn duration for each maneuver.

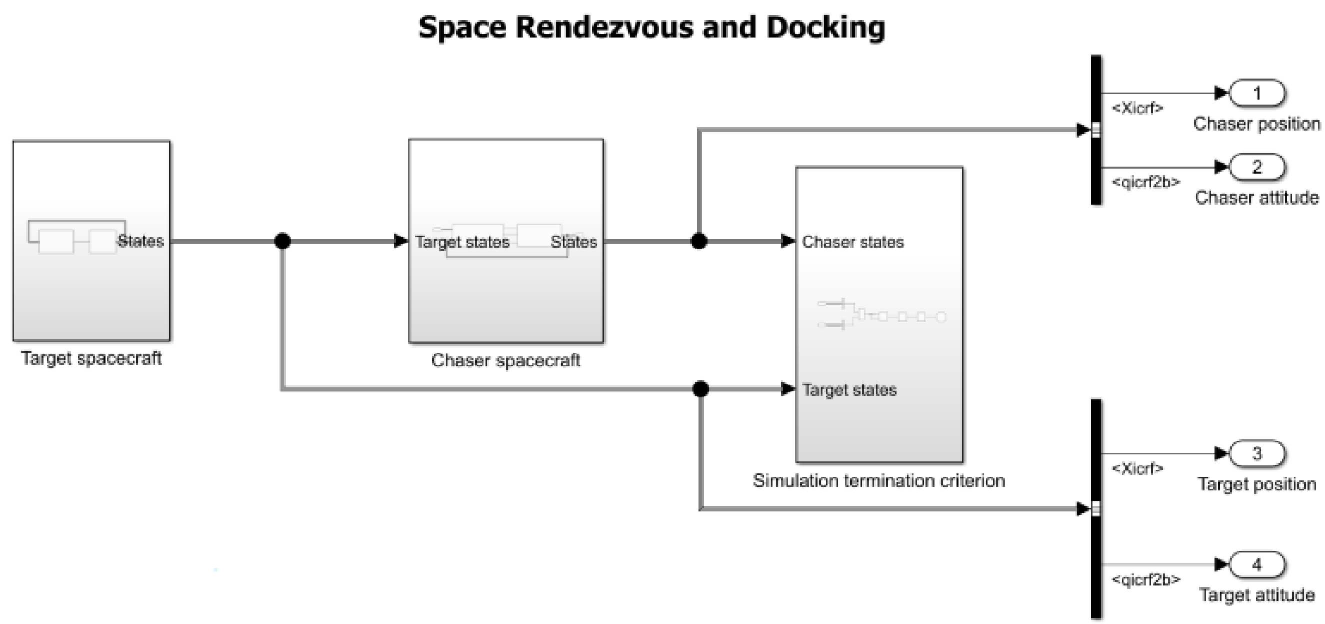
```
burnDuration1 = (m0-m1)/mDot;
burnDuration2 = (m1-m2)/mDot;
```

Mission Simulation

Open the simulation model.

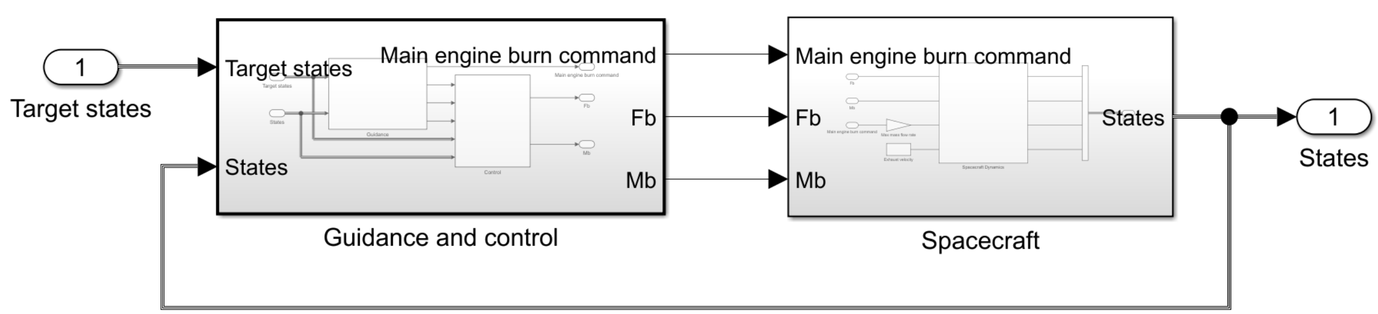
```
model = 'SpaceRendezvousAndDockingExample.slx';
open_system(model);
```

The top-level model consists of the `Target spacecraft`, `Chaser spacecraft`, and `Simulation termination criterion` subsystem. The `Chaser spacecraft` and `Target spacecraft` subsystems implement spacecraft dynamics and their respective guidance and control systems. The `Simulation termination criterion` subsystem ends the simulation when the distance between the chaser and the target is 1 m.



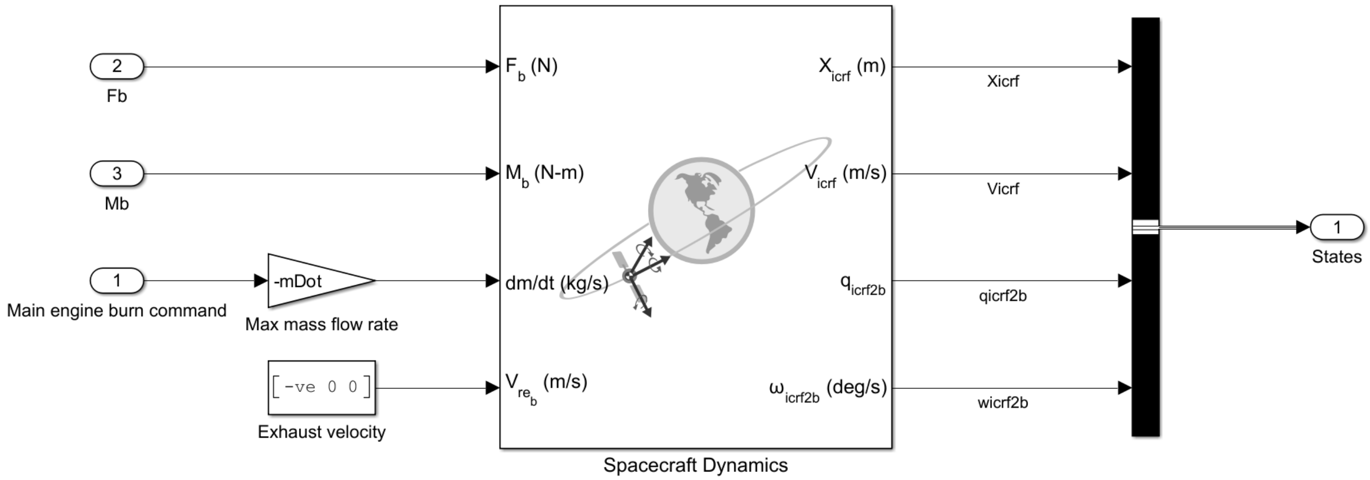
Both `Chaser spacecraft` and `Target spacecraft` subsystems consist of a `Guidance and control` subsystem and a `Spacecraft` subsystem. The `Guidance and control` subsystem of the chaser provides:

- Burn commands for the main propulsion system during the rendezvous phase
- Body force commands that the reaction control thrusters provide during the rendezvous phase
- Body moment commands that the reaction control thrusters provide to orient the chaser



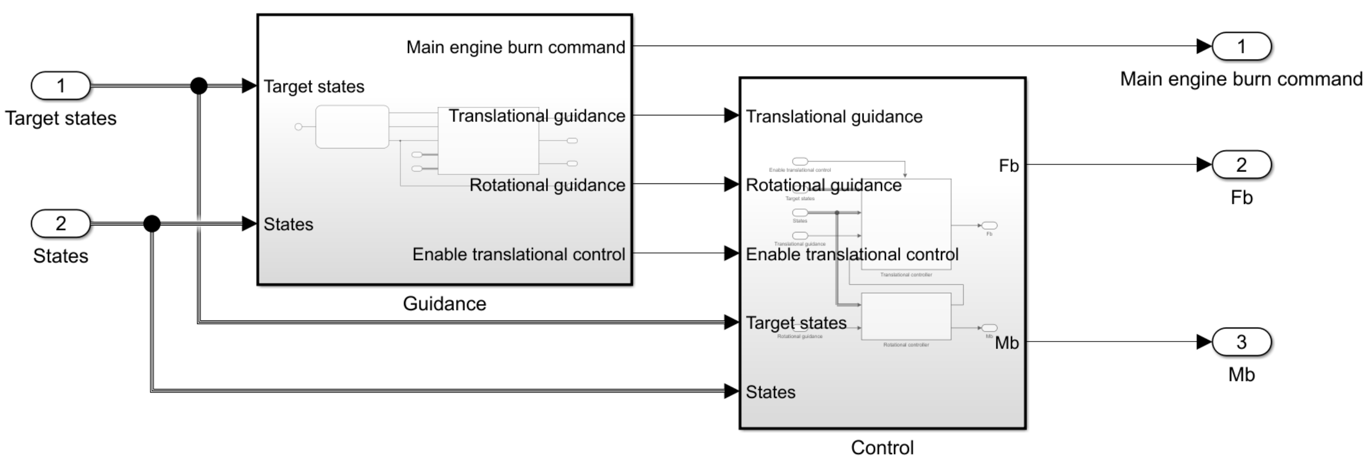
The `Guidance and control` subsystem of the target only provides body moment commands to the `Spacecraft` subsystem, as illustrated below. This is because the target does not perform any maneuvers and only maintains a constant attitude with respect to ICRF.

The `Spacecraft` subsystems of the chaser and the target implement the spacecraft dynamics using the Spacecraft Dynamics block, which accepts the body moment inputs. The `Spacecraft` subsystem of the chaser also accepts body force, mass flow rate, and exhaust velocity inputs, as the image shows.

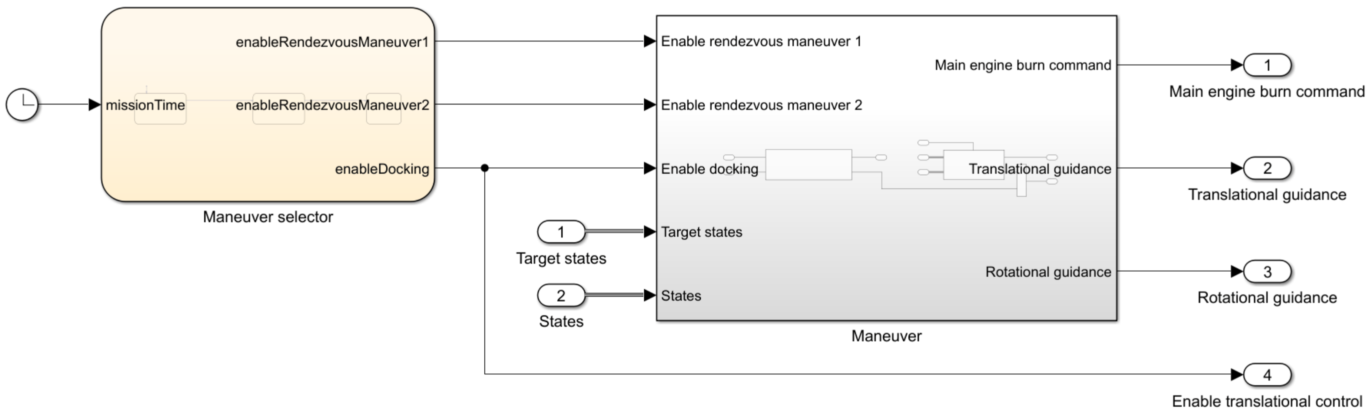


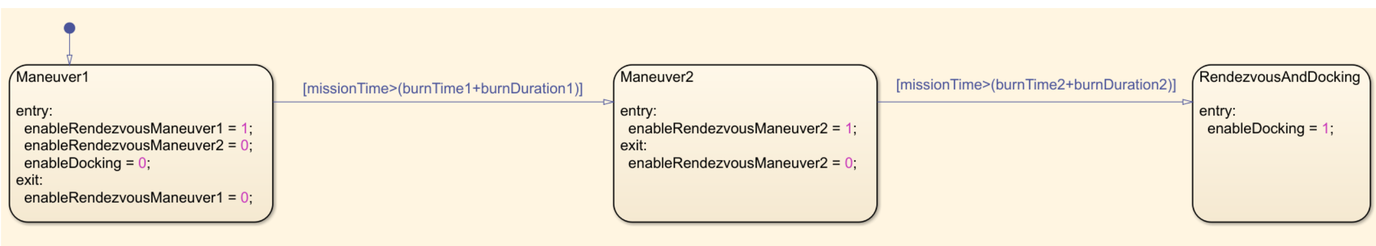
The mass flow rate and exhaust velocity inputs represent the main propulsion system, in which the mass changes when burning the engine. The body force and moment inputs represent the net thrust and moment that the reaction control thrusters must produce. The software ignores the mass change resulting from firing the thrusters.

The `Guidance and control` subsystems of the chaser and the target implement the guidance and control logic. Because the target does not perform any maneuvers and maintains a constant attitude with respect to ICRF, its `Guidance and control` subsystem only implements the rotational guidance and control systems. The `Guidance and control` subsystem of the chaser implements both translational and rotational guidance and control logics and issues the burn commands for the main propulsion system, as the image shows.

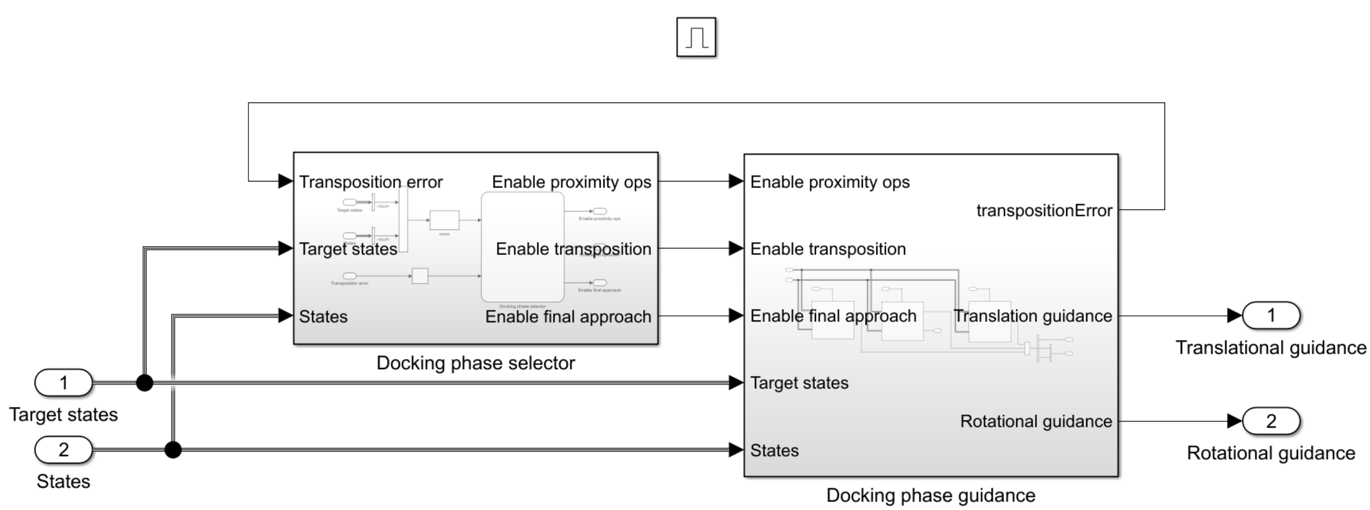


The `Guidance` subsystem issues burn commands during the rendezvous phase and translational guidance commands during the docking phase. The subsystem also provides rotational guidance throughout the mission. The translational guidance consists of the commanded relative velocity of the target with respect to the chaser in the body frame of the chaser. The rotational guidance consists of the commanded primary and secondary constraint vectors. The `Guidance` subsystem uses a Stateflow chart called `Maneuver selector` to implement the logic to determine which rendezvous maneuver must be executed and when to activate the docking phase, as the image shows.

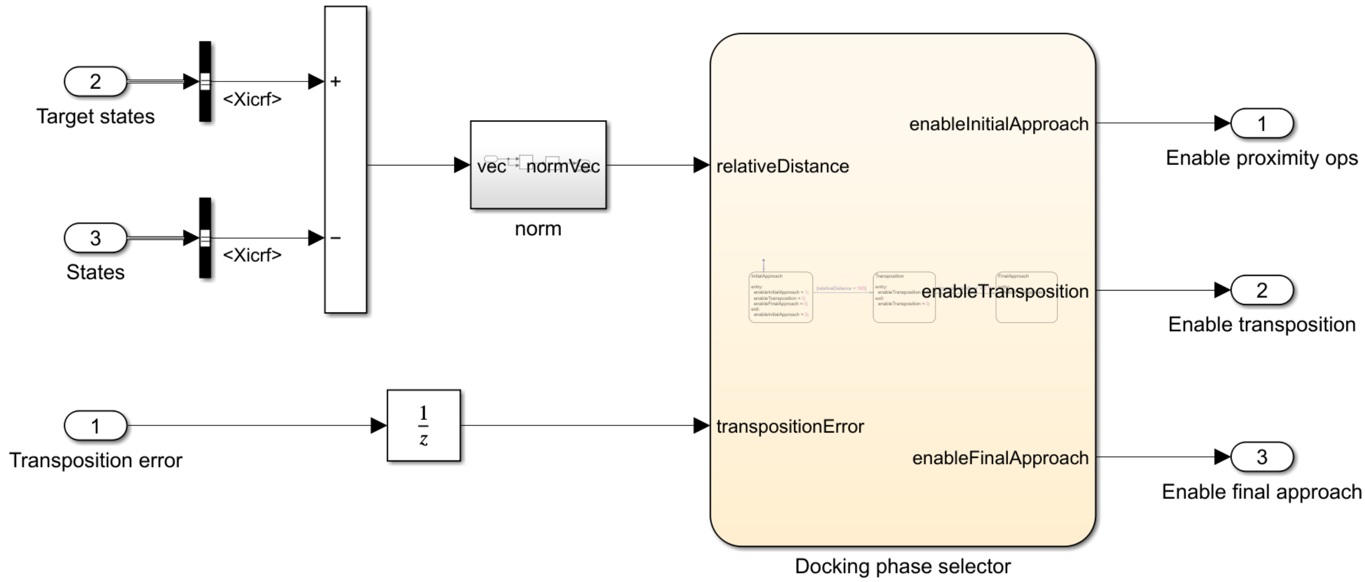


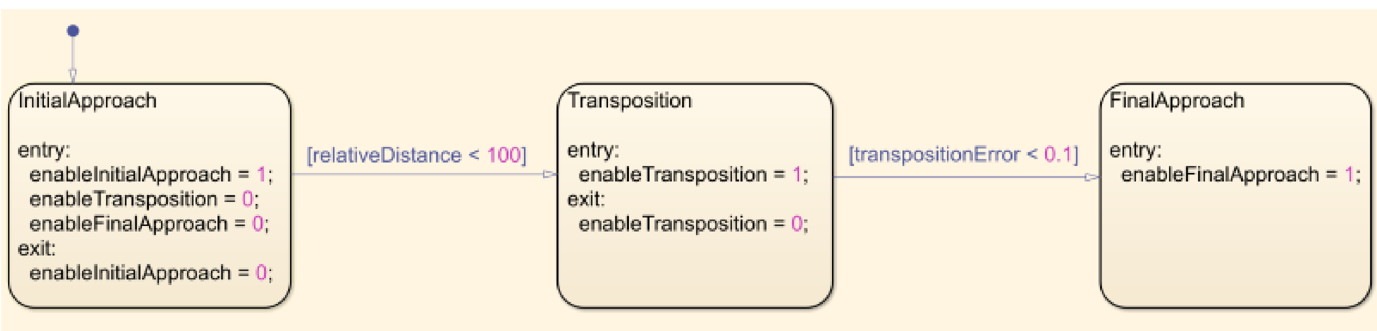


The `Maneuver` subsystem implements the guidance logic during the rendezvous and docking phases using the `Rendezvous` subsystem and the `Docking` subsystem, respectively. The `Docking` subsystem consists of the `Docking phase selector` and `Docking phase guidance` subsystems, as the image shows.

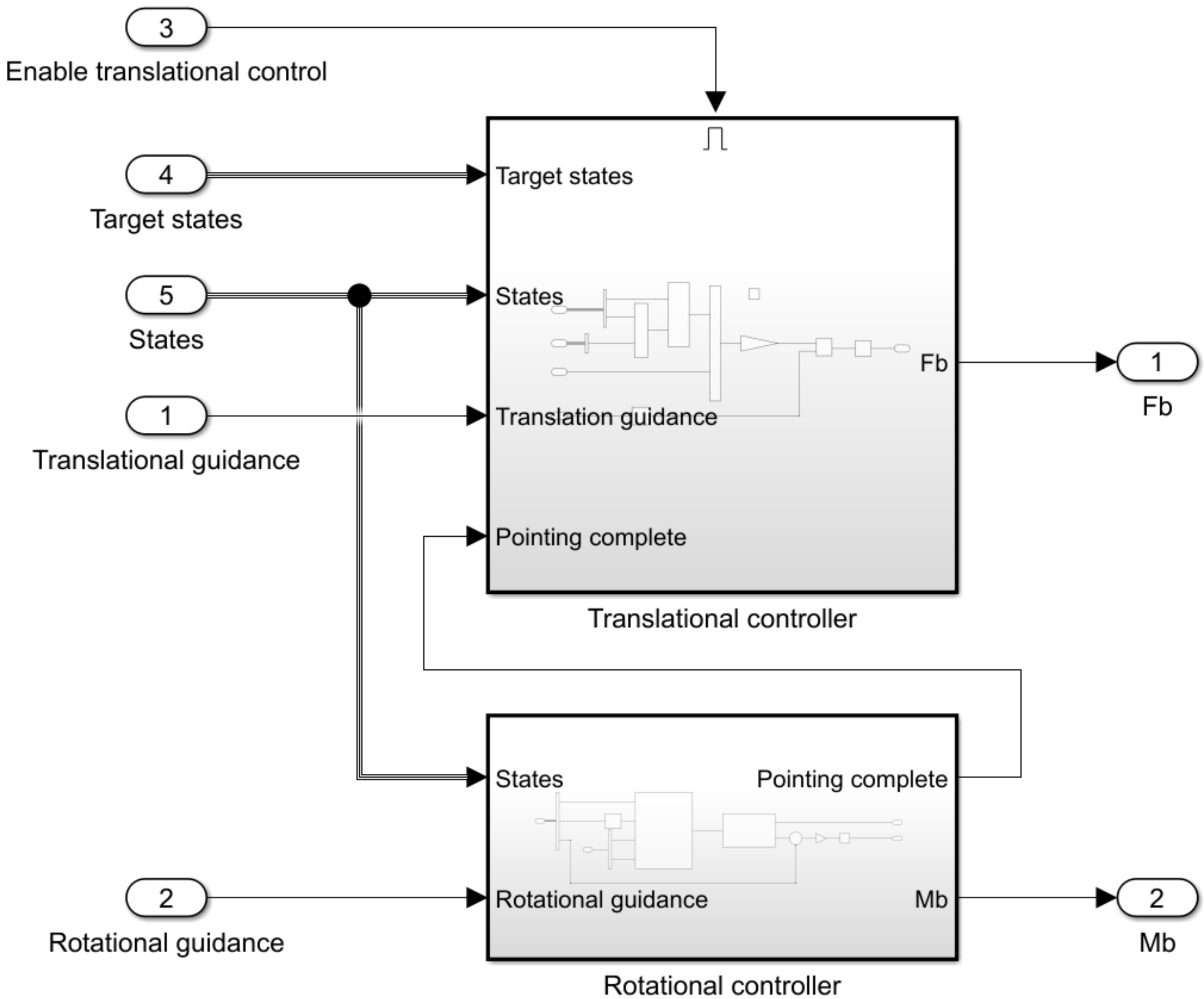


The `Docking phase guidance` subsystem implements the guidance logic for the different docking phases. The `Docking phase selector` subsystem uses a Stateflow chart called `Docking phase selector` to implement the logic to select the appropriate docking phase, as the image shows.

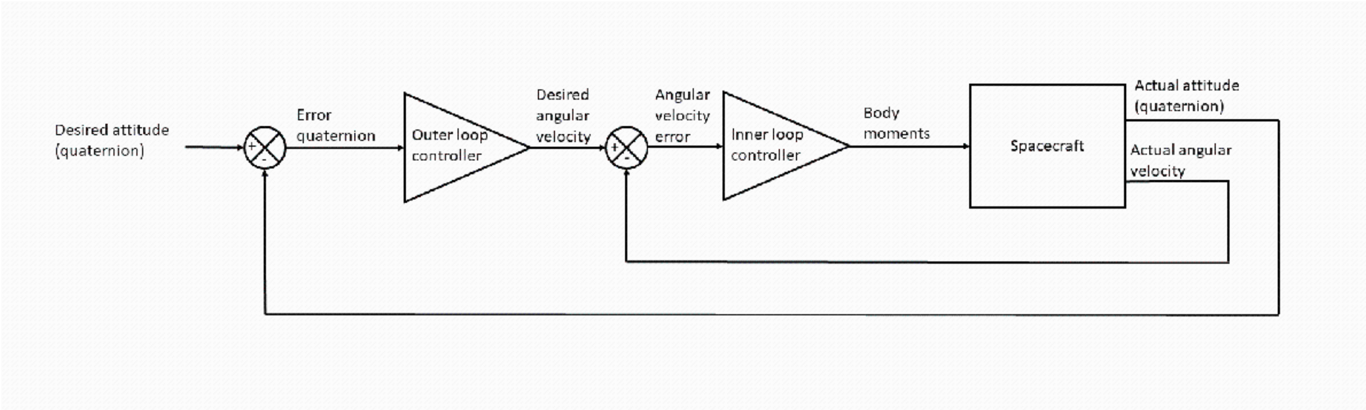




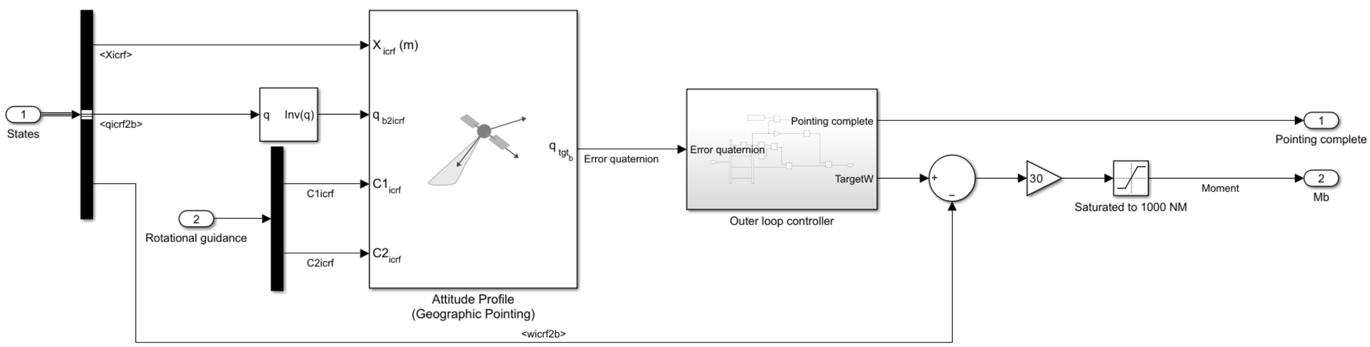
The guidance commands are fed into the control system. The target spacecraft contains a rotational controller that is implemented by the `Rotational controller` subsystem. The chaser spacecraft consists of a translational and a rotational controller, which are implemented by the `Control` subsystem, as the image shows.



The `Translational controller` subsystem is only active during the docking phase. It accepts the translational guidance commands and calculates the body force that the reaction control system must produce using a proportional controller. The `Rotational controller` subsystem accepts the rotational guidance commands and calculates the body moments that the reaction control system must provide. The `Rotational controller` architecture of the chaser and the target is the same and represents a PD controller, as the image shows.



The `Rotational controller` subsystem of the chaser implements this architecture.



The Attitude Profile block calculates the error quaternion based on the current attitude of the spacecraft and the desired attitude deduced from the rotational guidance commands, which in turn consist of the primary and secondary constraint vectors.

Specify a default end time for the simulation. Because the stop time is unknown ahead of the simulation, it is arbitrarily set to 100,000 s after the second burn time. The assumption is that 100,000 s provides sufficient time to complete the docking phase. The actual simulation end time will be lower than this value if the stopping criterion is met sooner.

```
endTime = burnTime2 + 100000;
```

Simulate the model. The model is configured to run in rapid accelerator mode for performance.

```
simOut = sim('SpaceRendezvousAndDockingExample.slx');
```

```
Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 7.1315s
```

Visualize Mission in Satellite Scenario Viewer

The top-level Simulink model sends the ICRF position and attitude of the chaser and the target to the workspace in `timeseries` format. Use this data to set up a satellite scenario and visualize the mission. Create a satellite scenario object. Set the start and stop times to those of the mission. Set the sample time such that there are 33,301 time samples in the scenario, which roughly translates to a scenario sample time of 2.5 s.

```
startTime = epoch;
stopTime = startTime + seconds(simOut.tout(end));
sampleTime = seconds(stopTime - startTime)/33300;
sc = satelliteScenario(startTime,stopTime,sampleTime)
```

```
sc = 
  satelliteScenario with properties:

         StartTime: 06-Aug-2022
          StopTime: 06-Aug-2022 23:07:26
        SampleTime: 2.4999
      AutoSimulate: 1
        Satellites: [1×0 matlabshared.satellitescenario.Satellite]
    GroundStations: [1×0 matlabshared.satellitescenario.GroundStation]
           Viewers: [0×0 matlabshared.satellitescenario.Viewer]
          AutoShow: 1
```

Extract the ICRF position and attitude history of the chaser and target.

```
chaserPos = simOut.yout{1}.Values;
chaserAtt = simOut.yout{2}.Values;
targetPos = simOut.yout{3}.Values;
targetAtt = simOut.yout{4}.Values;
```

Add the chaser spacecraft and define its attitude profile. Add a conical sensor 0.5 m in front of the chaser to mimic the docking port of the chaser.

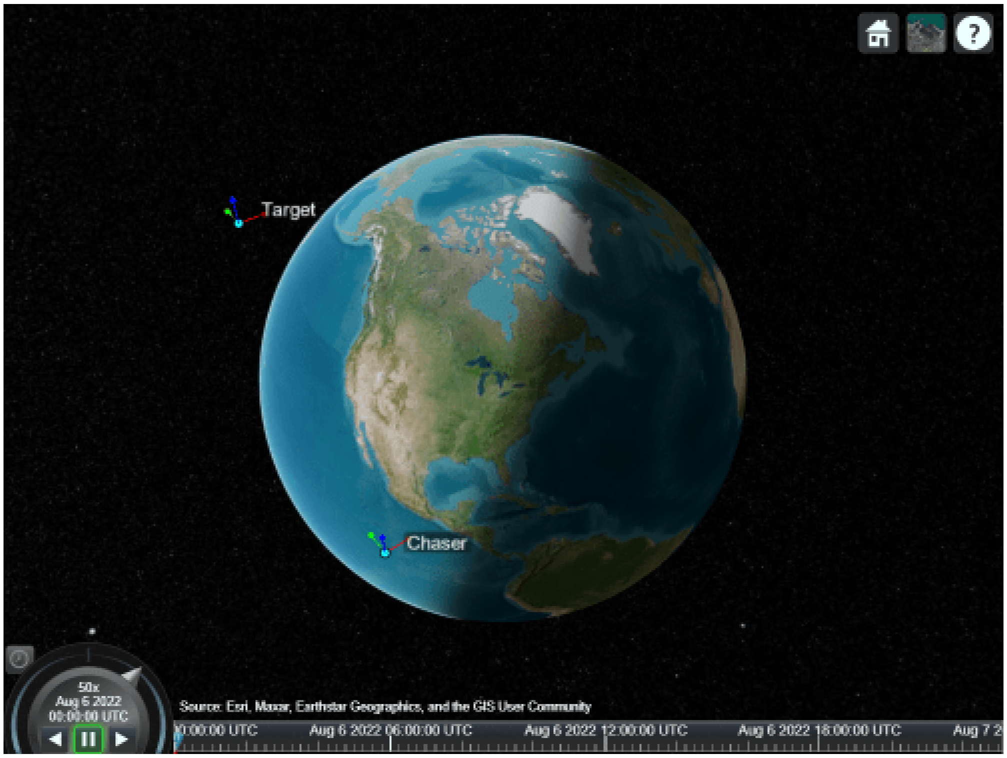
```
chaser = satellite(sc,chaserPos,Name='Chaser');
pointAt(chaser,chaserAtt);
chaserDockingPort = conicalSensor(chaser,MountingLocation=[0.5;0;0]);
```

Add the target spacecraft and define its attitude profile. Add a conical sensor 0.5 m in front of the target to mimic the docking port of the target.

```
target = satellite(sc,targetPos,Name='Target');
pointAt(target,targetAtt);
targetDockingPort = conicalSensor(target,MountingLocation=[0.5;0;0]);
```

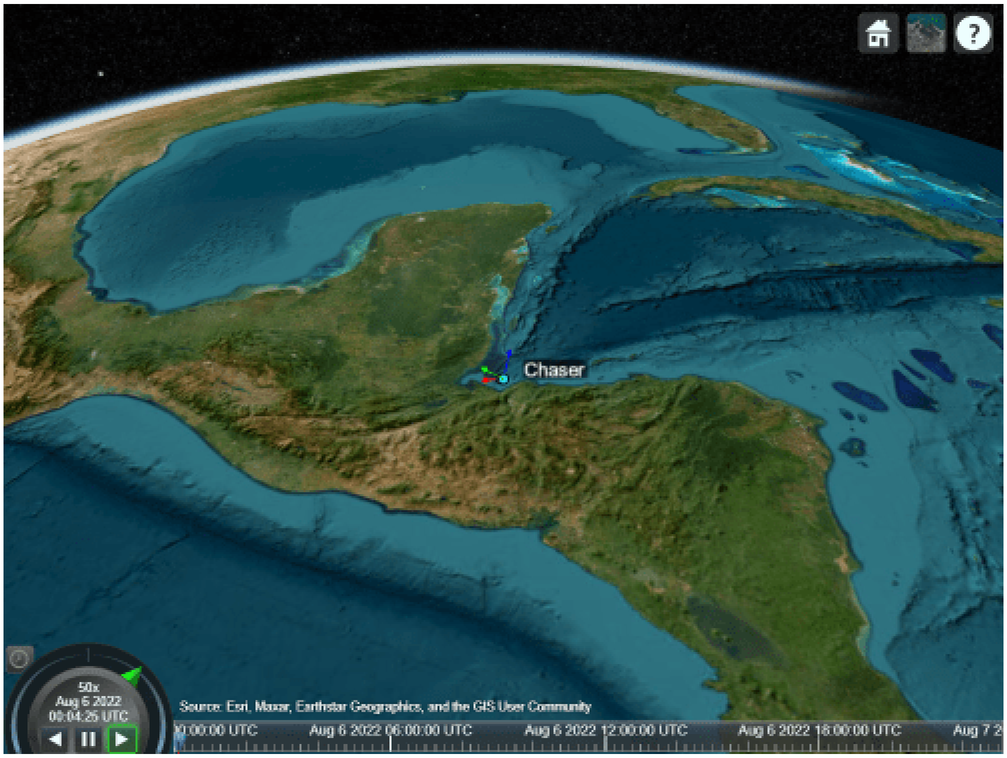
Launch a satellite scenario viewer and set the camera reference frame to `'Inertial'`. Visualize the coordinate axes of the chaser and the target. Hide the orbits of the chaser and the target.

```
v = satelliteScenarioViewer(sc,CameraReferenceFrame='Inertial');
coordinateAxes([chaser target]);
hide(chaser.Orbit);
hide(target.Orbit);
```

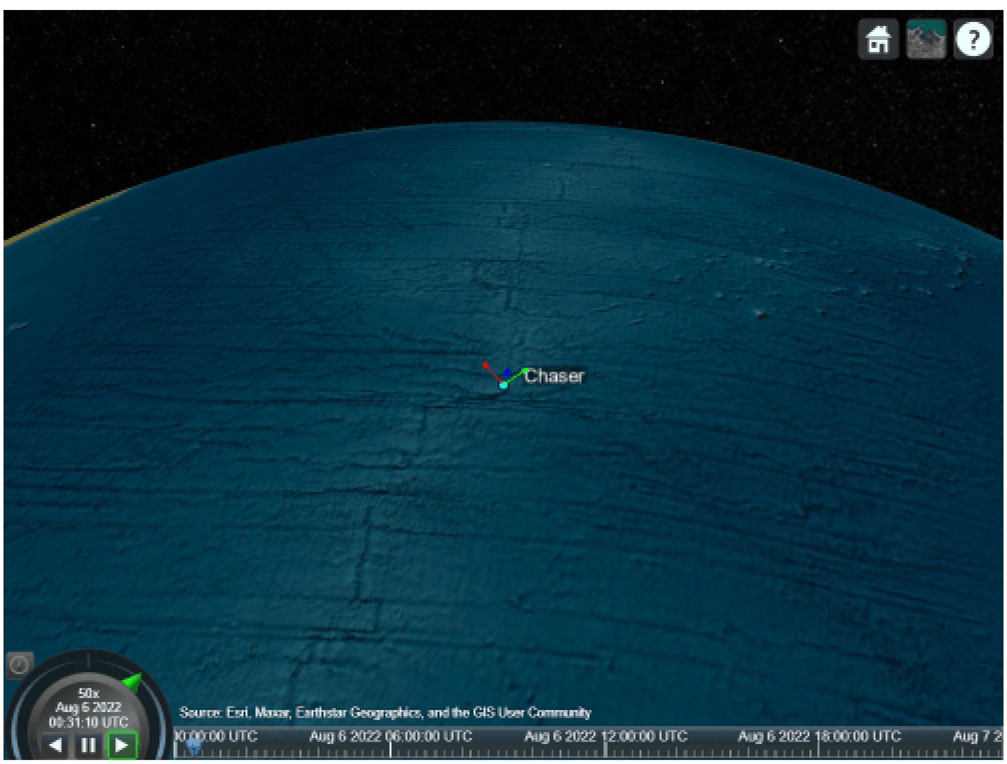


Play the scenario. Set the camera target to `chaser`. The chaser immediately reorients for the first rendezvous maneuver that is performed at about August 6, 2022, 12:24:16 AM UTC.

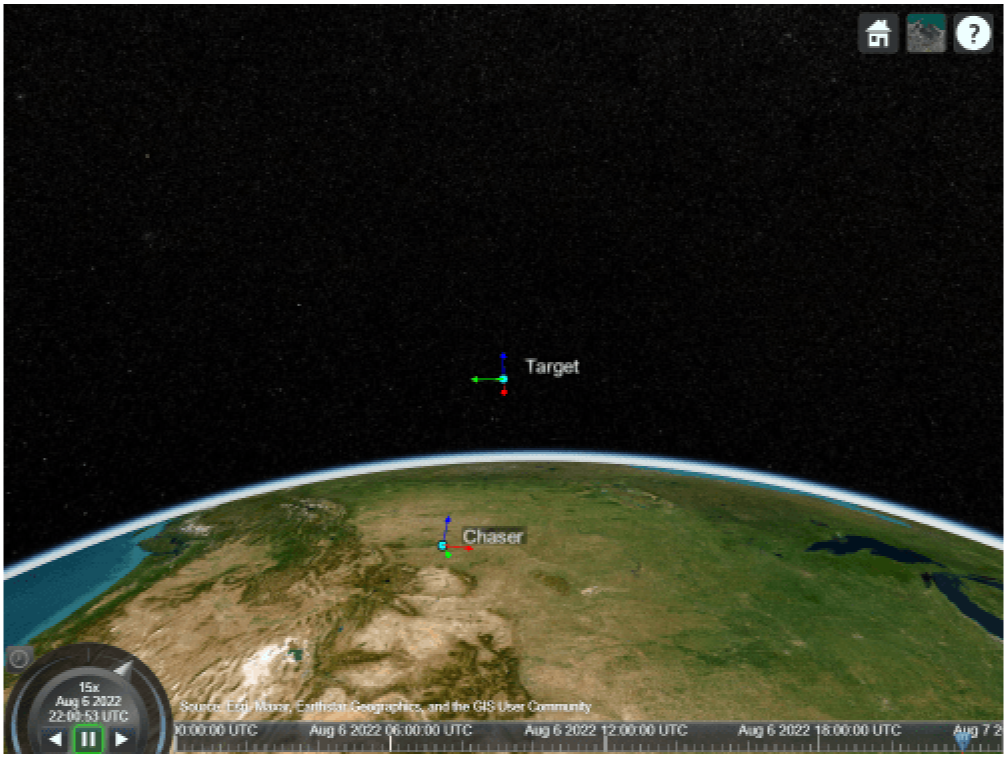
```
play(sc);
camtarget(v,chaser);
```



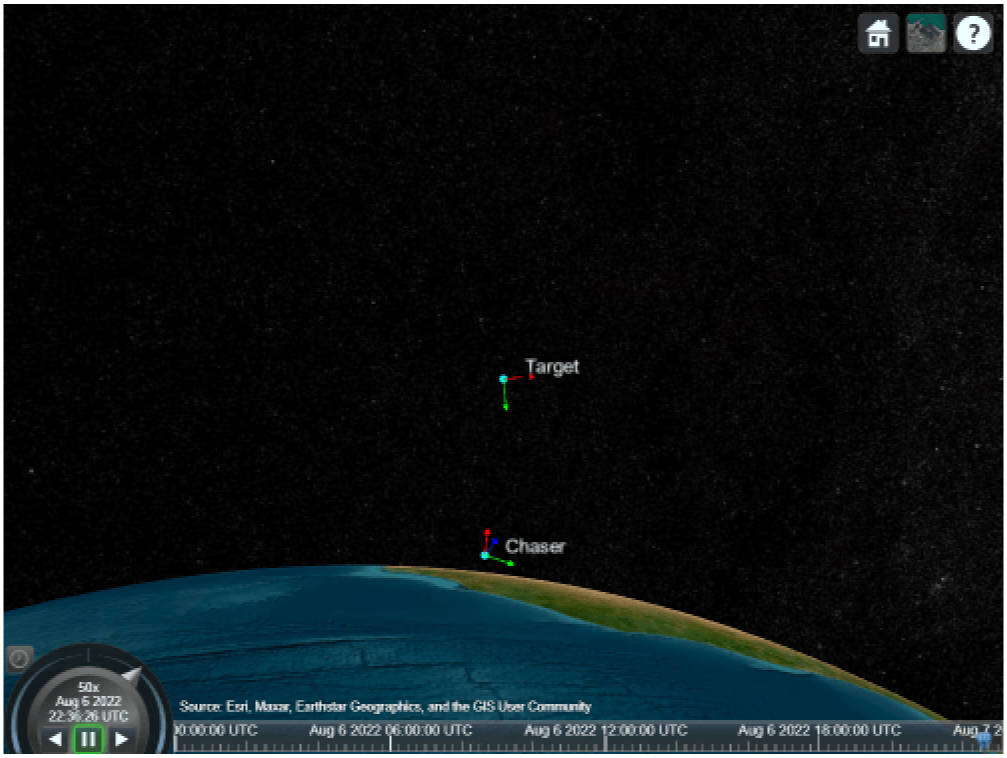
After the first maneuver is complete, the chaser reorients for the second rendezvous maneuver that is performed at about August 6, 2022, 10:00:49 PM UTC.



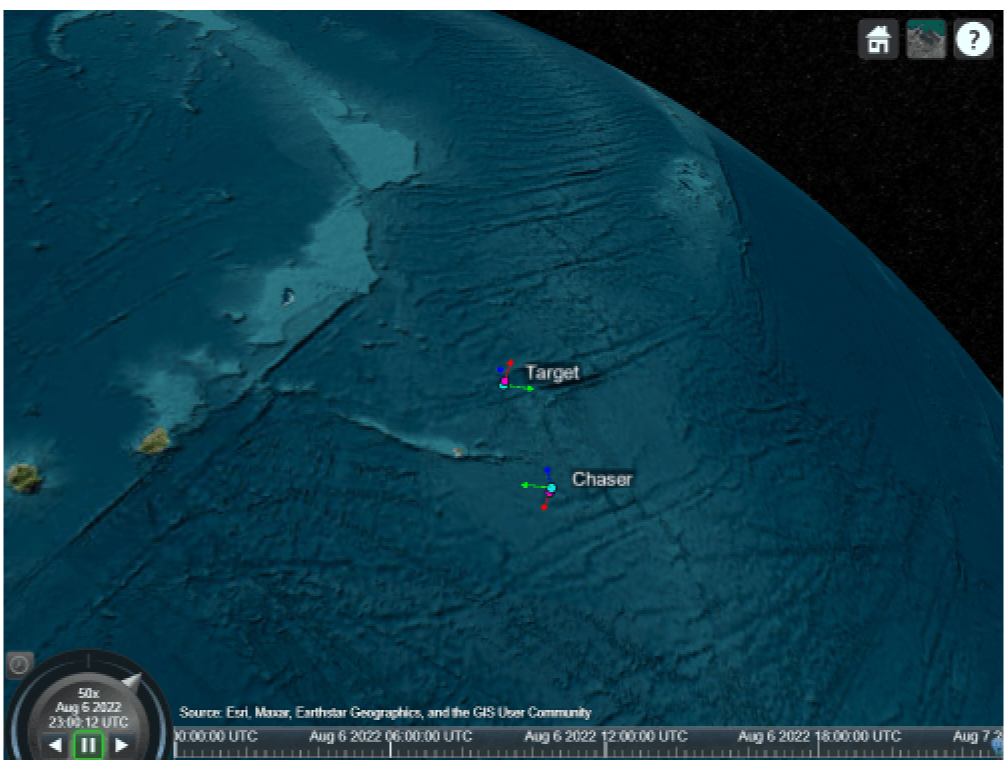
After the second rendezvous maneuver, the chaser is about 1 km below the target, with a small offset from the target radial position vector, and a relative velocity close to 0 m/s. The error in the position and velocity is the result of the approximation of the impulsive delta-v in the two rendezvous maneuvers.



After rendezvous is complete, the chaser enters the docking phase. It reorients for initial approach attitude. Its translational reaction control thrusters eliminate the lateral offset from the target radial position vector, and it starts closing in on the target at 0.3 m/s.

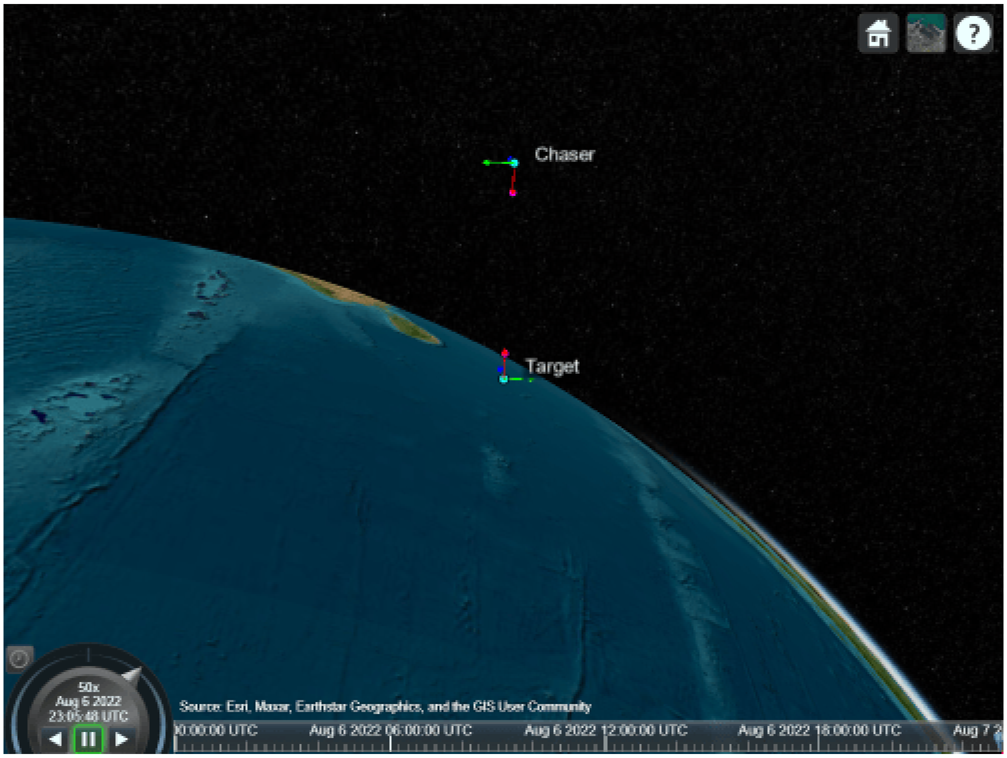


After the distance between the chaser and the target reduces to 100 m, the chaser enters the transposition phase. It reorients to the final docking attitude and maneuvers around target to position itself 20 m in front of the target.



Once transposition is complete, the chaser enters the final approach phase, in which the chaser starts closing in on the target at 0.3 m/s.

When the range to target reduces to 10 m, the chaser reduces its closing rate to 0.03 m/s.



Docking is complete when the range to target further reduces to 1 m.

Modeling and Simulation of an Autonomous Underwater Vehicle

This example shows how to simulate the dynamics and control of an autonomous submarine using Simulink® and the Aerospace Blockset™. Given a reference position/velocity, this example designs a control law enabling the autonomous submarine to obey the reference parameters.

Examples of the use of autonomous underwater vehicles include deep-sea geological surveying, exploration, and search and rescue. Autonomizing these vehicles saves the weight of a human crew, reducing the overall mission cost and increasing the scope of mission objectives (for example, enabling large sample returns or enabling more fuel to be stored on board, which can lead to a longer mission duration).

For the purposes of this example, all motion is performed via propulsion, without the use of control surfaces.

This example uses this workflow of the autonomous submarine - given a set of reference parameters (either a target position or velocity), and sensor-measured values of the vehicle's current velocity and position, the controller calculates appropriate thrust commands to pass to the plant for actuation. After the plant implements a thruster control input, sensors pass information about the new position and velocity of the submarine to a noise filter and then to the controller to determine the next set of thrust commands required for the vehicle to obey the reference parameters.

Modeling Assumptions

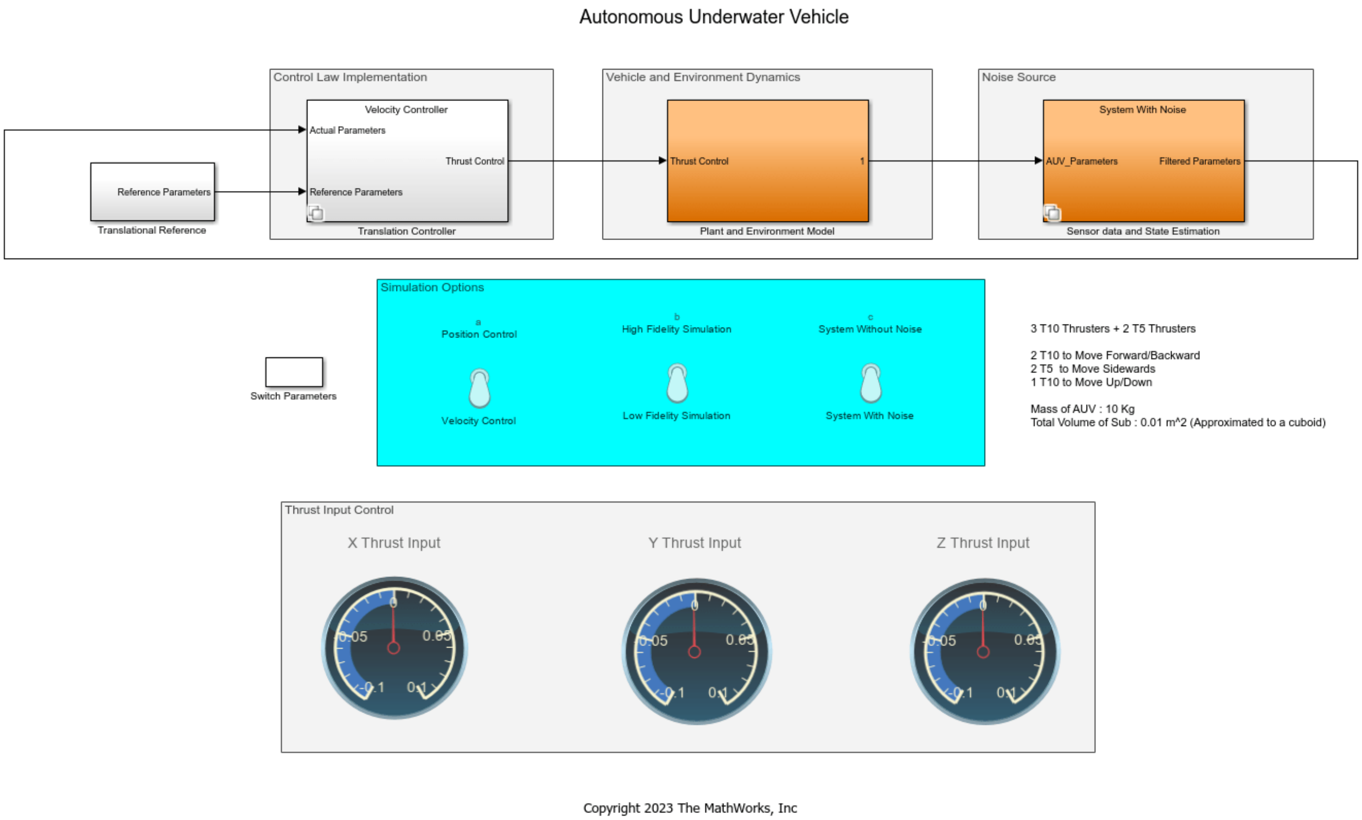
Assume that the submarine is cuboidal in shape, with a mass of 10 kg and a volume of 0.01 m^3. For underwater propulsion, the vehicle uses thrusters of two different sizes, rated at 5 and 10 pound-force of thrust (T5 and T10). In total, the vehicle uses five thrusters arranged as follows: - Two T10 Thrusters to allow for forward/backward motion, referred to as surge. - Two T5 Thrusters to allow for sideways motion, referred to as sway. - One T10 Thruster to allow for upward/downward motion, referred to as heave.

Assume that all thrusters in a group operate identically. For example, the two T10 thrusters intended to provide surge always provide identical amounts of thrust at any given instant of time. This means that they cannot be individually commanded to produce different thrusts. In addition, this simulation does not model any weight changes.

Assume that the submarine is always fully submerged in the water. This means that the vehicle does not resurface and therefore no component of the vehicle interacts with air above the water.

Open the Model

```
open_system("asbAUV");
```



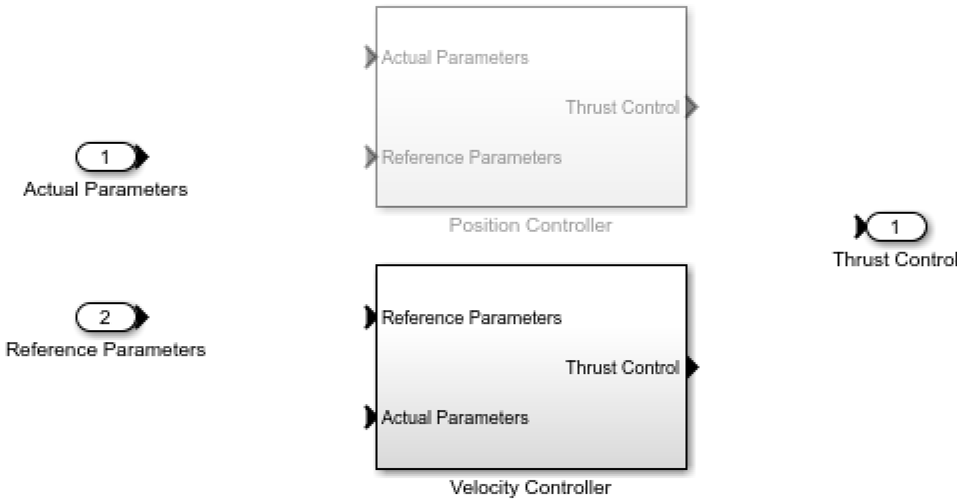
The top level diagram closely resembles the high-level workflow presented in the introduction.

The model presents three simulation options: - The type of control, in this case position or velocity. - The fidelity of the simulation with respect to the modeling of the environmental forces. A high fidelity simulation involves a more accurate computation of the lift and drag generated by the submarine. A low fidelity involves a less accurate computation. - The presence of noise in the sensor measurements.

Control Strategy

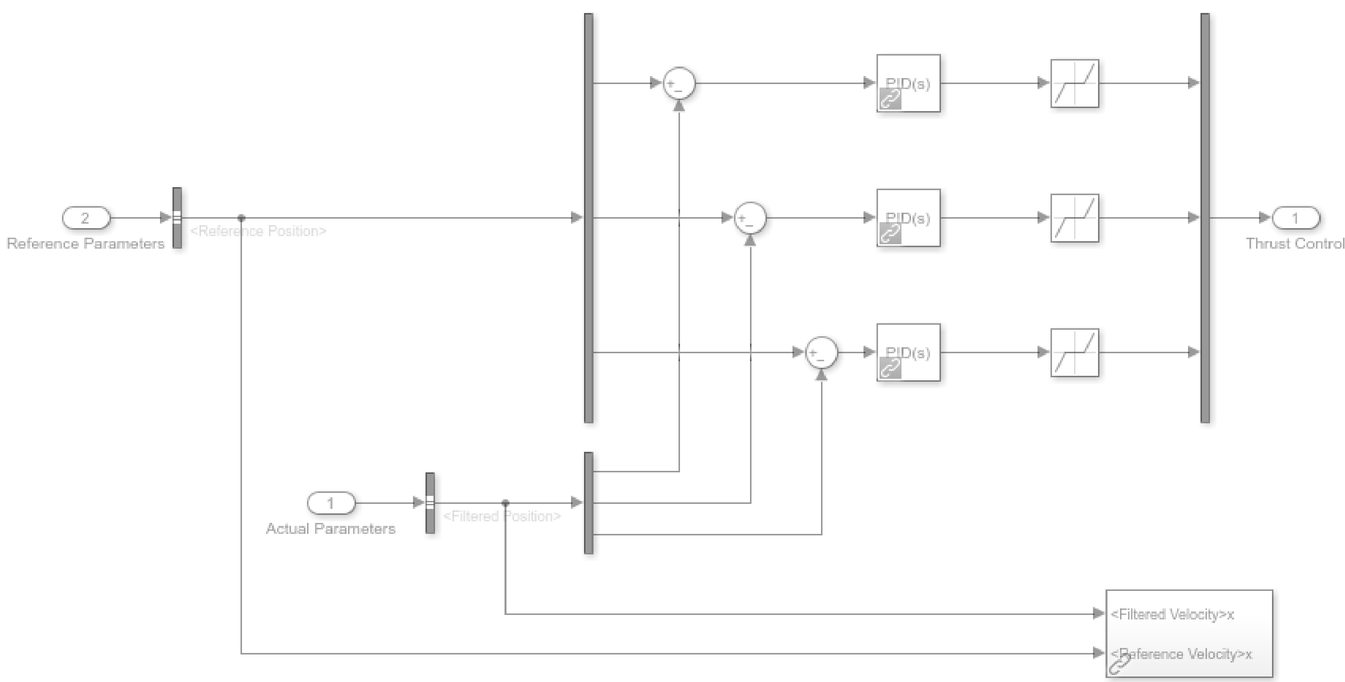
The "Translational Reference" subsystem contains the reference parameters for the vehicle to follow. The Signal Editor block generates these parameters. The model passes these parameters and the sensor measurements of the vehicle's actual position and velocity to the translational controller.

```
open_system("asbAUV/Translation Controller")
```



Depending on the selection in the top-level diagram, either the position controller or the velocity controller is activated. The "Position Controller" subsystem uses one PID controller block to determine the required thrust command in each direction, requiring three controller blocks in total.

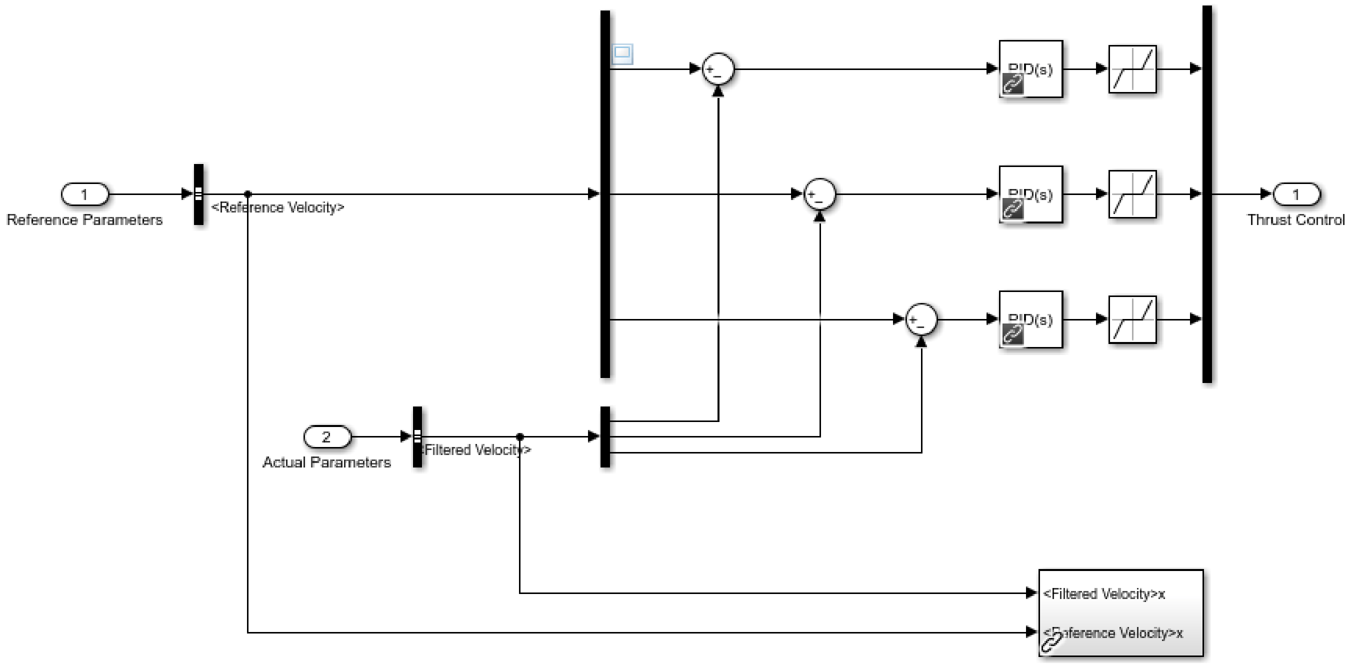
```
open_system("asbAUV/Translation Controller/Position Controller")
```



The resulting thrust commands are then passed through dead zone blocks to discard values of thrust control that are less than 0.05.

The "Velocity Controller" subsystem also uses three PID controller blocks to determine the required thrust commands.

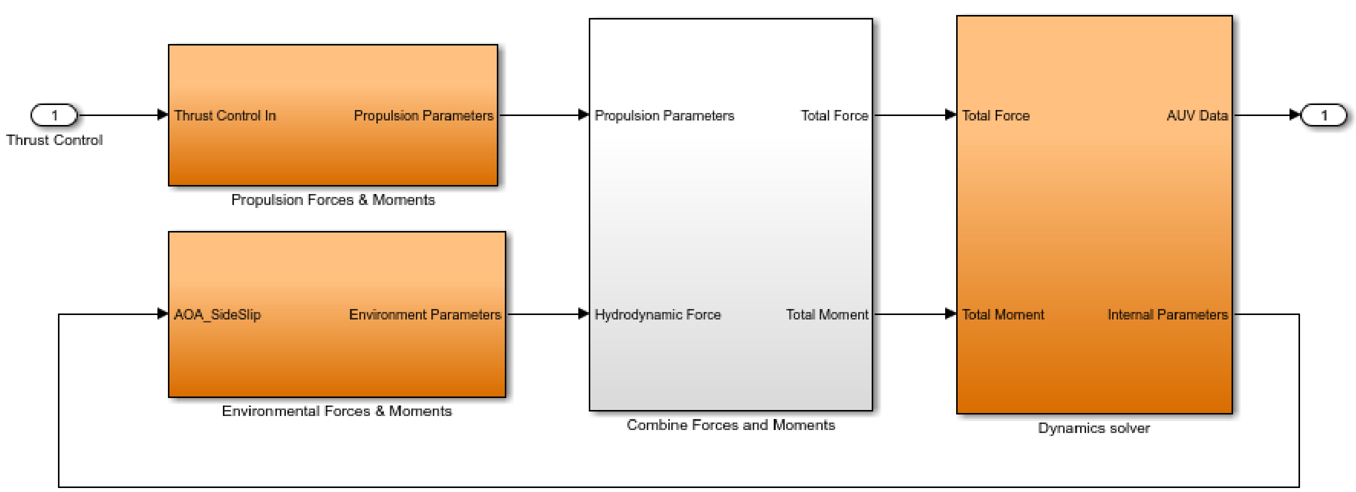
```
open_system("asbAUV/Translation Controller/Velocity Controller")
```



Vehicle and Environment Modeling

The computed thrust commands are then passed to the "Plant and Environment Model" subsystem.

```
open_system("asbAUV/Plant and Environment Model")
```

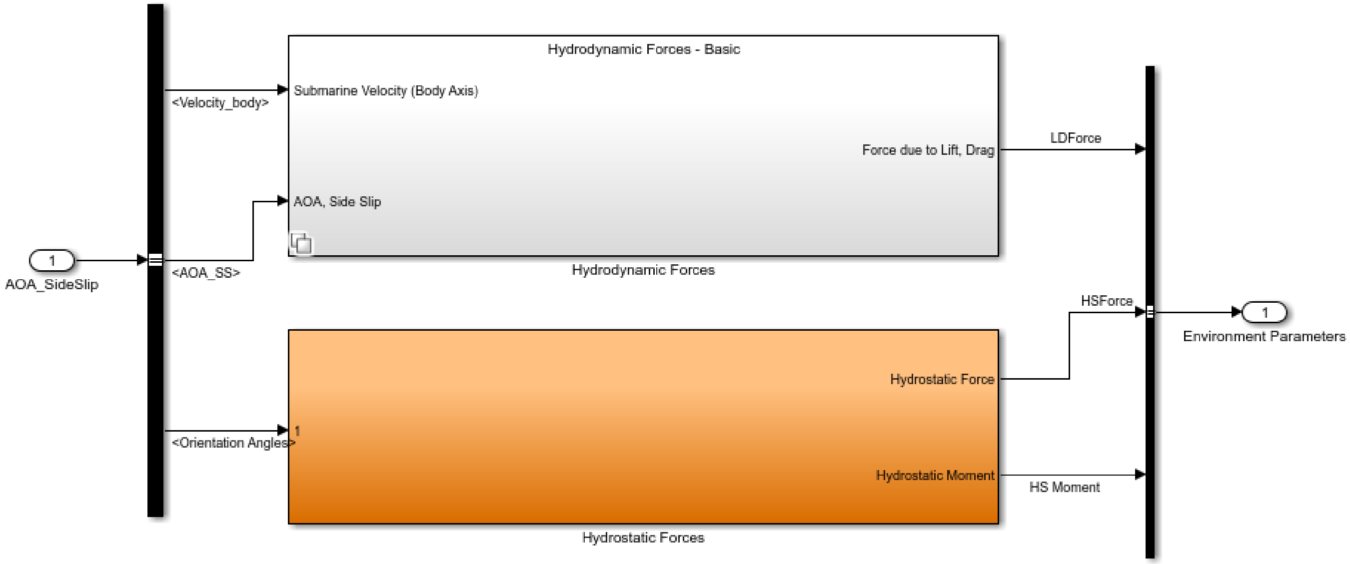


In the "Plant and Environment Model" subsystem, the computed thrust commands are first passed to the "Propulsion Forces and Moments" subsystem, which converts these thrust commands into Pulse-Width Modulated (PWM) signals using a 1D Lookup Table block. The subsystem then converts the PWM signals into resultant thrust forces using the transfer functions of the thrusters.

These resultant thrust forces are then used to compute the resulting moments on the vehicle.

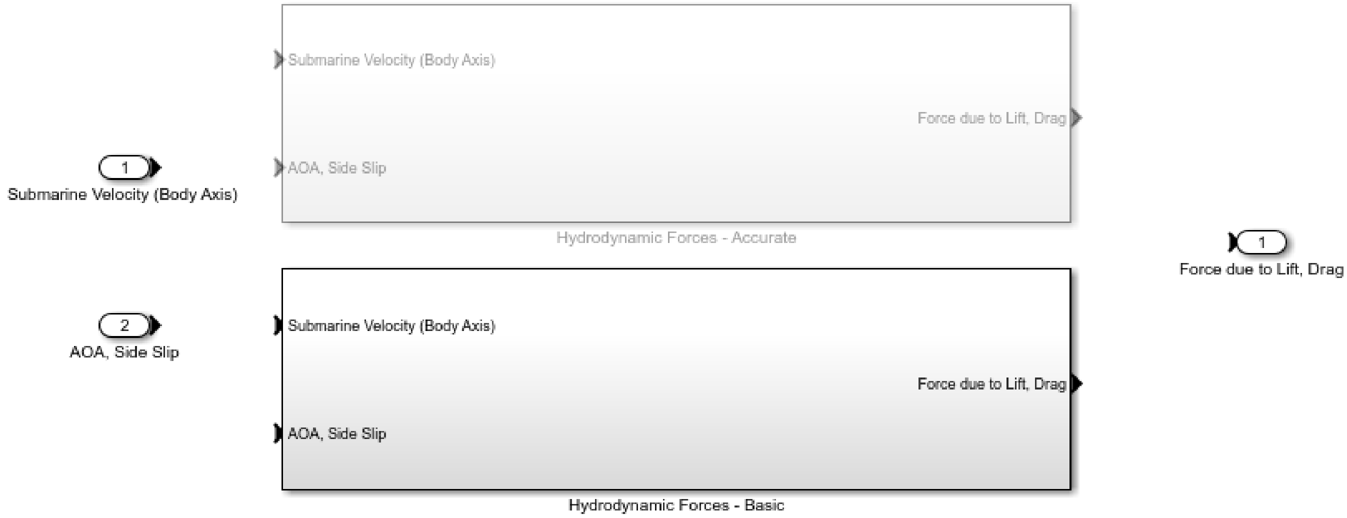
To compute the hydrodynamic (lift and drag) and hydrostatic (weight and buoyancy) forces, the "Environmental Forces and Moments" subsystem accepts the vehicle angle of attack, sideslip angle, velocity in the body frame, and the Euler orientation angles relative to the inertial frame as inputs.

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments")
```



The "Hydrodynamic Forces" subsystem contains two conditionally operated subsystems, "Hydrodynamic Forces - Accurate" and "Hydrodynamic Forces - Basic".

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments/Hydrodynamic Force
```



The selected level of simulation fidelity determines the subsystem that gets activated.

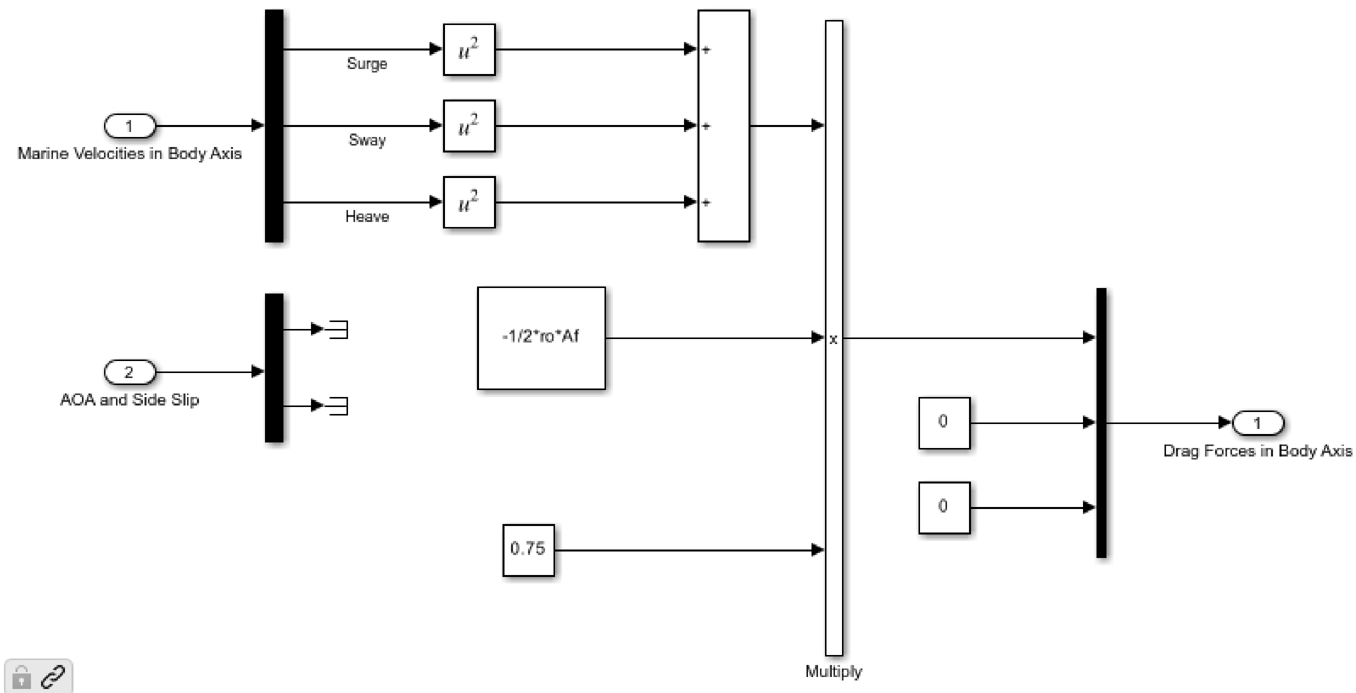
- High Fidelity Simulation: The "Hydrodynamic Forces - Accurate" subsystem becomes activated and the computation of lift and drag takes into account the effects caused by the angle of attack and the sideslip angle on the classic lift/drag equation.

- Low Fidelity Simulation: The "Hydrodynamic Forces - Basic" subsystem becomes activated and the computations of lift and drag do not consider these effects and only use the classic lift/drag equations.

Regardless of the chosen level of simulation fidelity, each subsystem performs lift and drag computations in separate subsystems.

In the "Hydrodynamic Forces - Basic" subsystem, the computation of drag assumes a drag coefficient, CD, of 0.75. Note that the signals for angle of attack and sideslip angles are ignored.

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments/Hydrodynamic Force
```



The net result is given by:

$$F_{drag} = \frac{1}{2}\rho V^2 A_{frontal} C_D$$

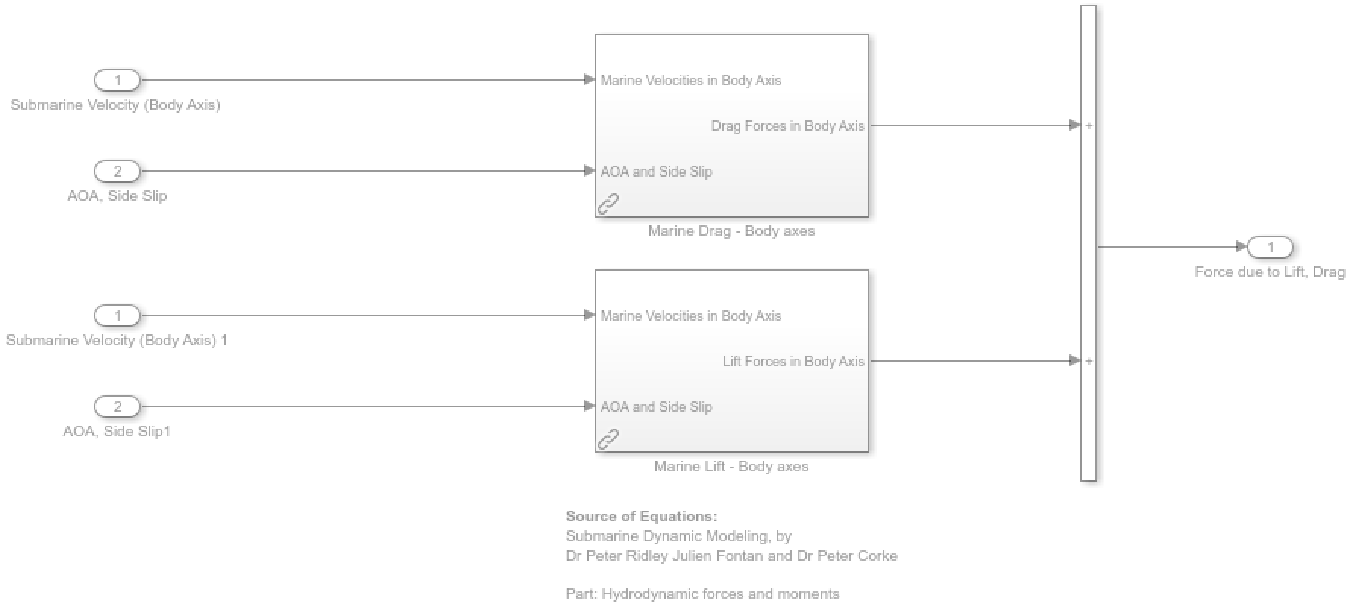
Where: - A_frontal represents the frontal area, assumed to be 0.001 m^2. - V represents the vehicle's velocity. - rho represents the density of the water. - CD represents the coefficient of drag.

The computation of lift is performed using the same equation with a lift coefficient of 0.2 and assuming the same frontal area.

$$F_{lift} = \frac{1}{2}\rho V^2 A_{frontal} C_L$$

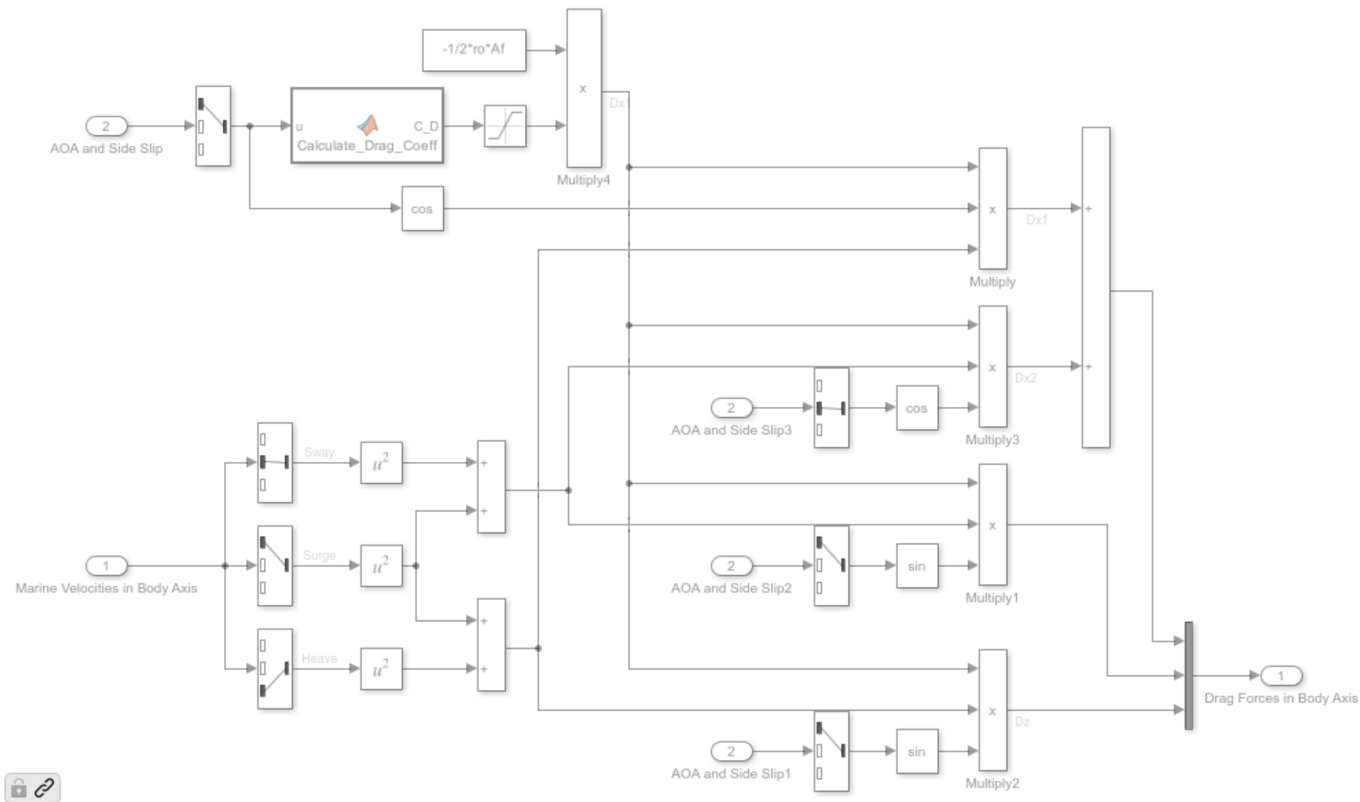
If a high-fidelity simulation is selected, the "Hydrodynamic Forces - Accurate" subsystem becomes activated. This selection also componentizes the computation of lift and drag into their own subsystems.

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments/" + ...
    "Hydrodynamic Forces/Hydrodynamic Forces - Accurate")
```



Note that in the "Force Exerted due to Drag (Body axis)" subsystem, the angle of attack and sideslip angles are now used.

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments" + ...
    "/Hydrodynamic Forces/Hydrodynamic Forces - Accurate/Marine Drag - Body axes")
```



Letting alpha represent the angle of attack, beta represent the side-slip angle, Vx, Vy, and Vz represent surge, sway, and heave respectively, drag is computed as:

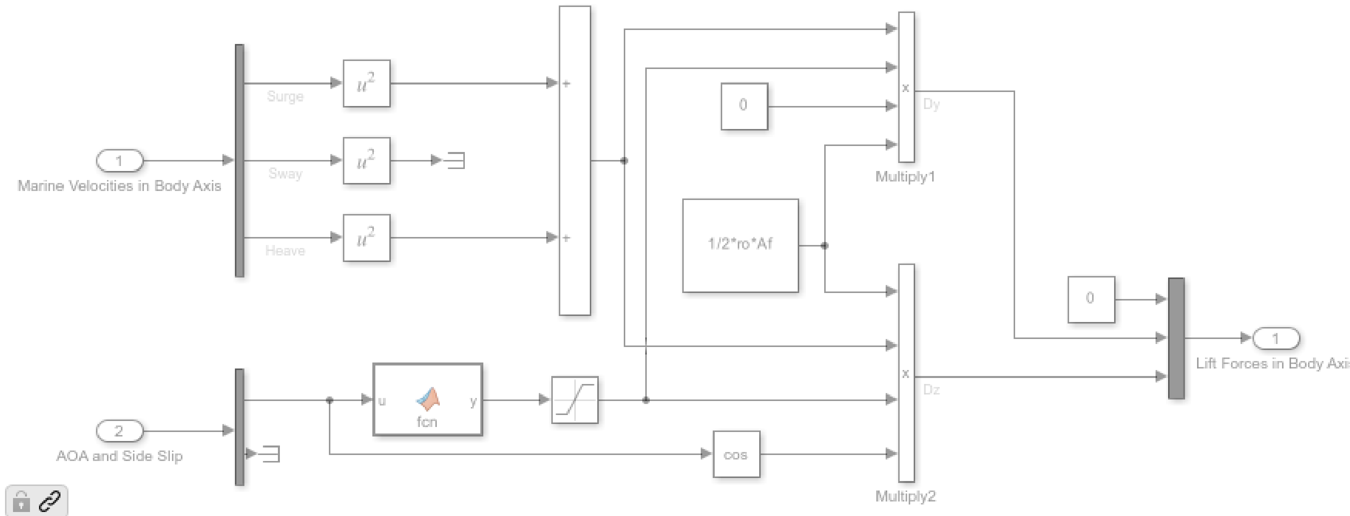
$$F_{drag} = \frac{1}{2}\rho A_f C_D[(V_x^2 + V_y^{2)})\sin(\beta) + (V_x^2 + V_z^2)\sin(\alpha) + (V_y^2 + V_z^2)\cos(\alpha) + (V_x^2 + V_y^2)\cos(\beta)]$$

The drag coefficient is computed as a function of angle of alpha as follows.

$$C_D = 5.25\alpha^2 - 0.315\alpha + 0.732$$

Similarly, the accurate computation of lift also considers the effects of angle of attack. Side-slip angle does not affect the generated lift because the example is only concerned with the vertical component of force generated due to forward motion. This computation occurs in the "Force Exerted due to lift (Body axis)" subsystem.

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments/Hydrodynamic Forc
```



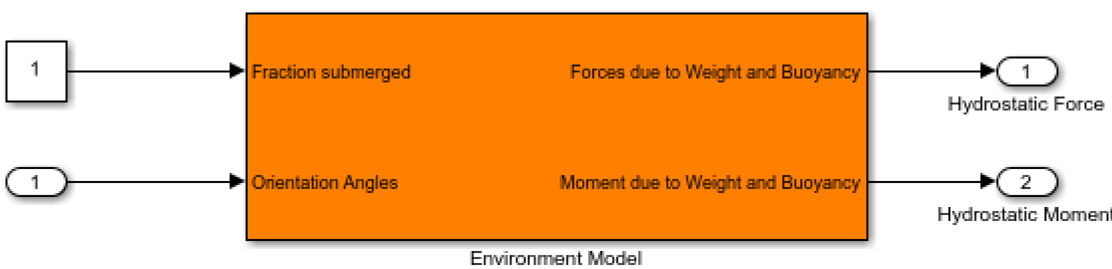
The net result is given by:

$$F_{lift} = \frac{1}{2}\rho A_f C_L (V_x^2 + V_z^2)\cos(\alpha)$$

For small angles, the coefficient of lift varies linearly with angle of attack.

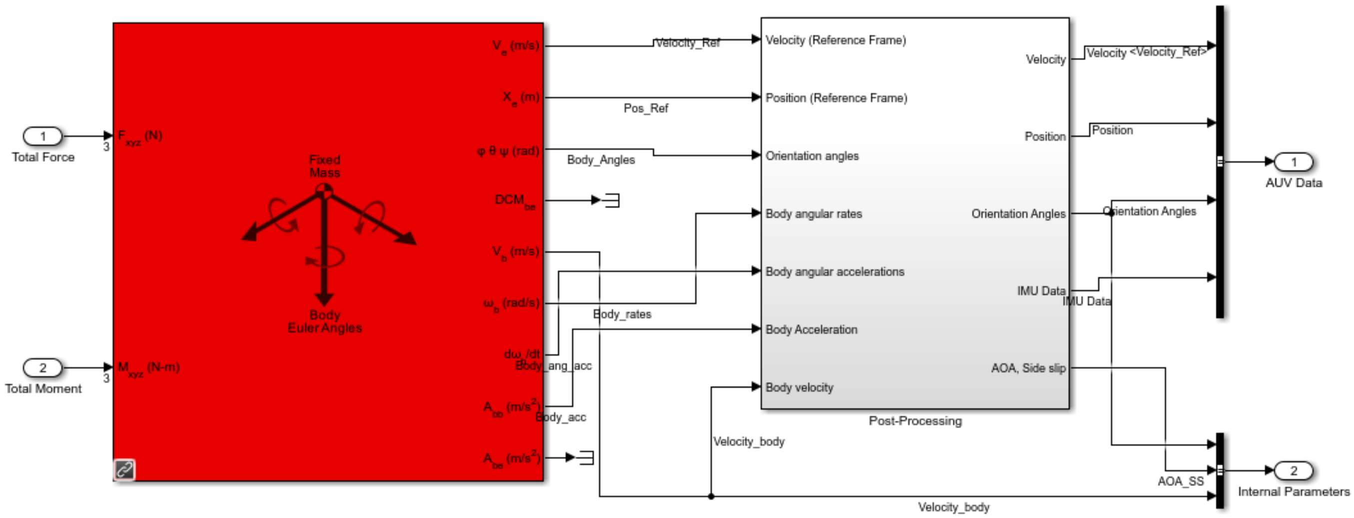
Simultaneously, the hydrostatic force and moment (due to the vehicle's weight on buoyancy) are computed inside the "Hydrostatic Forces" subsystem.

```
open_system("asbAUV/Plant and Environment Model/Environmental Forces & Moments/Hydrostatic Forces
```



The computed hydrodynamic, hydrostatic, and previously computed propulsion forces and moments are passed to the "Combine Forces and Moments" subsystem, which vectorially adds all the forces and moments to be passed to the "Dynamics Solver" subsystem, whose contents are pictured below. The subsystem accepts the total forces and moments as inputs. It outputs the vehicle velocity, position, angle of attack, side-slip angle, orientation angles, angular rates, and accelerations after applying the previously computed thrust input.

```
open_system("asbAUV/Plant and Environment Model/Dynamics solver")
```



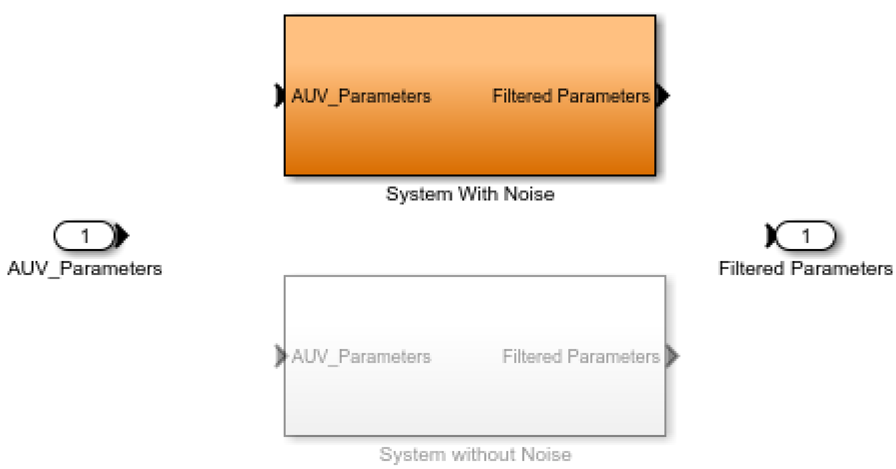
For organizational purposes, the velocity, position, angle of attack, side-slip angle, orientation angles, and their rates are grouped into a bus signal labelled "AUV Data". The orientation angles, angle of attack, side-slip angle, and velocity in the body frame are grouped into another bus signal called "Internal Parameters".

The AUV Data signal is passed onto the subsystem "Sensor Data and State Estimation" (see top level diagram), while the internal parameters are fed back to the "Environmental Forces and Moments" subsystem to compute the total force and moment resulting from the hydrodynamic and hydrostatic forces at the next time step.

Sensor Data and State Estimation

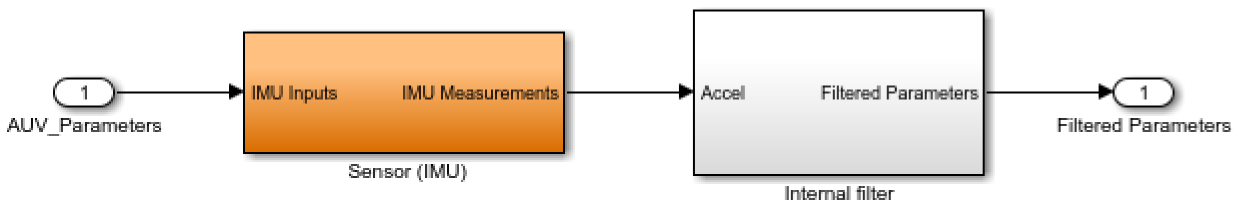
The aim of the "Sensor Data and State Estimation" subsystem is to mimic a sensor measurement and demonstrate the use of filters to remove noise by introducing noise into the vehicle's current motion parameters.

```
open_system("asbAUV/Sensor data and State Estimation")
```



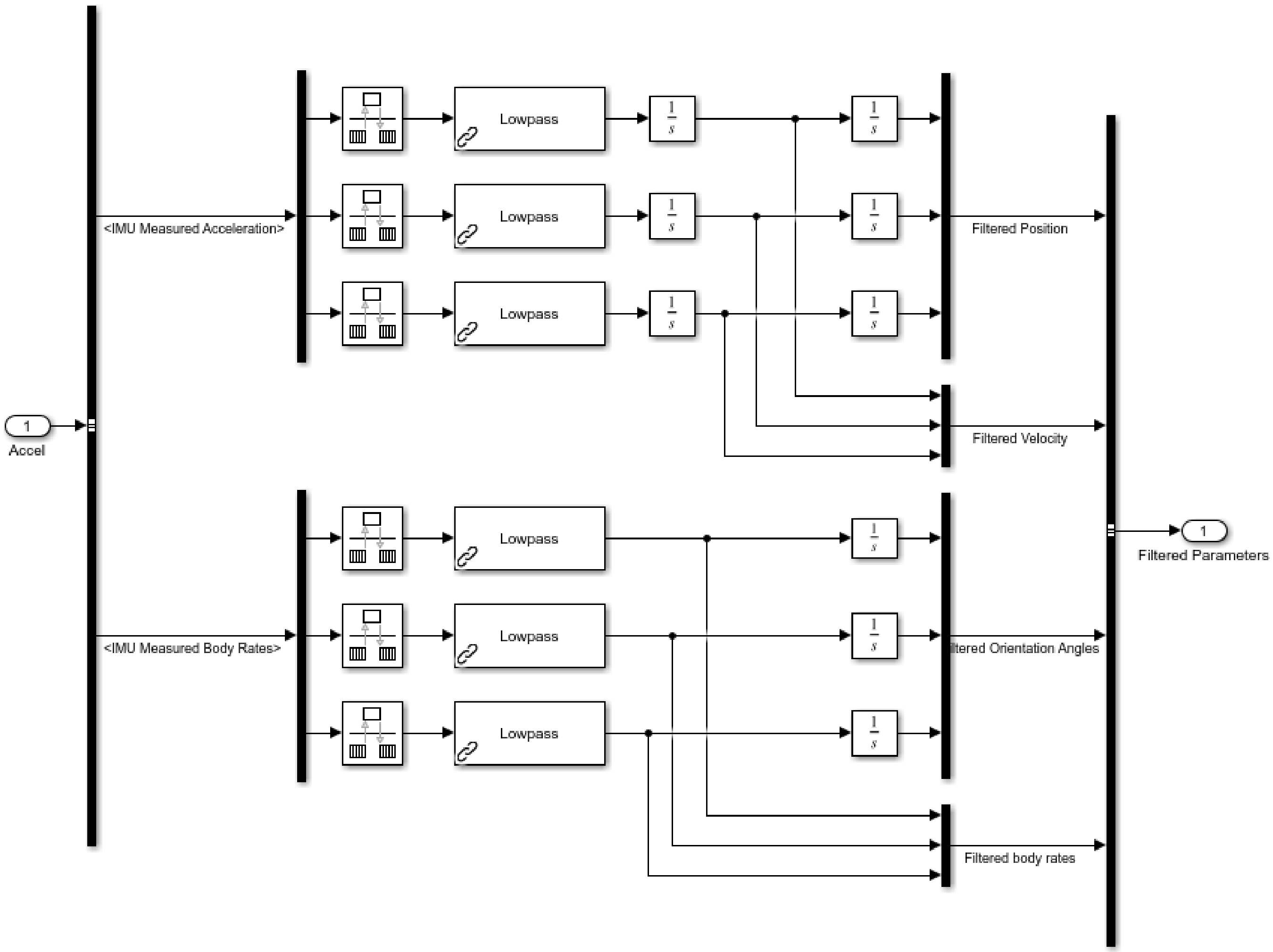
Your decision on the inclusion of noise is reflected in the subsystem that is activated. If you want to simulate without noise, the AUV parameters are passed through this subsystem with no modification. Conversely, if you decide to simulate with the presence of noise, the "System With Noise" subsystem is activated.

```
open_system("asbAUV/Sensor data and State Estimation/System With Noise");
```



The AUV data (comprising the vehicle's accelerations, angular rates, angular accelerations, and center of gravity location) and the gravitational constant are passed to the Three-axis Inertial Measurement Unit block, which generates noisy measurements of the vehicle acceleration and angular rates. A Mux block combines these signals into a single signal called "IMU Measurements", which is passed to the "Internal Filter" subsystem.

```
open_system("asbAUV/Sensor data and State Estimation/System With Noise/Internal filter")
```



The IMU Measurements signal is separated back into measured acceleration and measured angular rates. Each quantity has three components (x, y, and z) which are passed through a lowpass filter to obtain a best estimate of the relevant quantity. These components are integrated over time to obtain filtered quantities of the vehicle's position, velocity, orientation angles, and angular rates in the body frame.

These filtered quantities are then passed into the "Translation Controller" subsystem to compute the required thrust commands for the next time step.

Why use the Aerospace Blockset?

Hand-modeling six coupled equations of motion in Simulink is prone to errors. To reduce the chance of modeling errors arising from this, use the 6DOF block from the Aerospace Blockset. The 6DOF block computes the following position data, given the total force and moment acting on a body:

- Vehicle velocity in the flat Earth frame - Vehicle position in the flat Earth frame - Euler rotation angles - The direction cosine matrix relating the body frame to the flat Earth frame - Vehicle velocity in the body-fixed frame - Angular velocity of the vehicle in the body-fixed frame - Rate of change of angular velocity (angular acceleration) in the body- fixed frame - Translational acceleration in the body-fixed axes - Translational acceleration with respect to the inertial frame

The above outputs are in different reference frames, which may not be ideal for some applications. To ensure that quantities remain in desired reference frames, use the rotation and coordinate transformation blocks.

For convenience, convert Euler rotation angles to direction cosine matrix elements and vice versa. To achieve this, use the Direction Cosine Matrix to Rotation Angles and Rotation Angles to Direction Cosine Matrix blocks.

The Three-axis Inertial Measurement Unit block is useful in the state estimation part of the problem. Given the vehicle's actual acceleration, angular rates, angular accelerations, center of gravity location, and the acceleration due to gravity, the block mimics an IMU and outputs measured values of the vehicle's acceleration and angular rates.

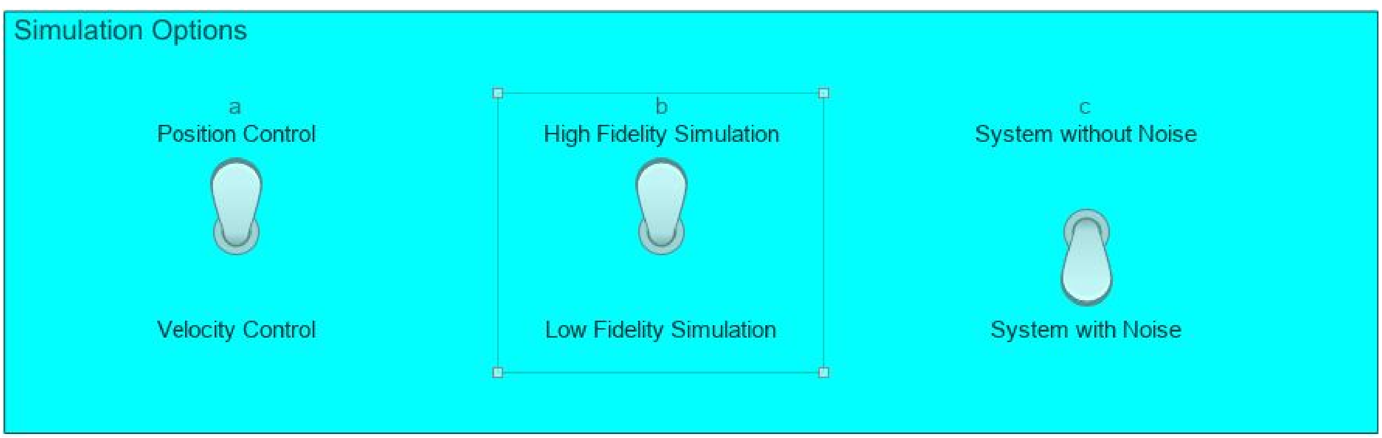
The Aerospace Blockset also lets you simulate a 6DOF system in which the vehicle's mass is variable. This capability lets you model fuel consumption as the simulation runs, resulting in a more accurate model. Other available functionalities include actuators, aerodynamics simulations, instrument modeling, center of gravity estimation, etc.

Simulation - Position Control

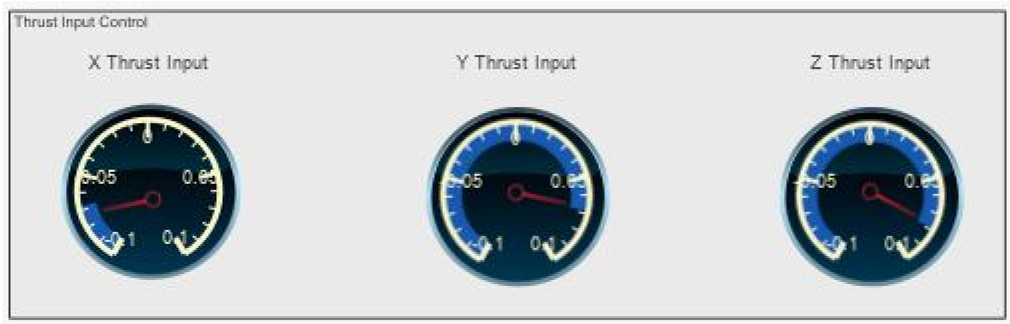
For the purposes of this simulation, the vehicle's position is controlled in a high-fidelity simulation with noise present in the measurements of the vehicle's current parameters. The simulation time is set at 300 seconds.

Looking at the reference parameters for position, the X and Y coordinates of the final position are specified to be -10. The Z coordinate of the reference position remains at zero until 150 seconds into the simulation, at which time it changes to 5 meters. This is implemented via a Step function block.

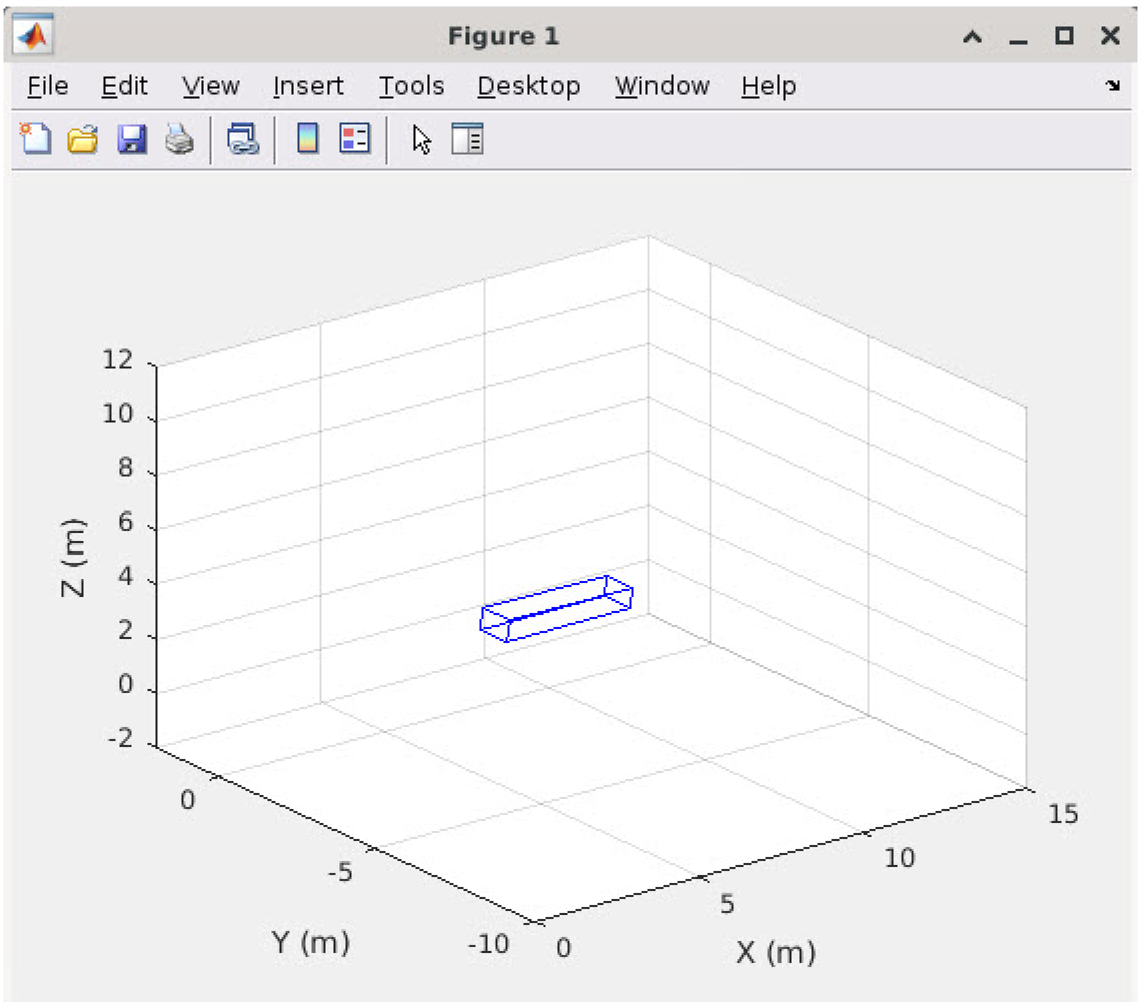
Start out by setting the switches to the desired positions.



Then, click the 'Run' button or press Ctrl+T. After compilation, the model starts to run and a figure window showing the live orientation and position of the vehicle is opened. At 150 seconds of simulation time, the vehicle can be seen translating upwards, consistent with the reference parameters for position provided in the "Translational Reference subsystem". Furthermore, the thrust inputs can be verified via the thrust input gauges located below the switches on the top-level diagram.



After the simulation completes, a second figure window opens which plots the trajectory history of the vehicle in three dimensions. From here, it can be verified that the vehicle's motion satisfies the reference parameters provided.



Alternate Control Strategies

The control problem illustrated by this example can be further extended to be an optimal control problem. In other words, as opposed to merely controlling the position and velocity of the vehicle, the controller must now control the position and velocity while simultaneously satisfying another set of constraints. For example, keeping the vehicle on a known path while minimizing fuel consumption. In view of this, two alternate control strategies can be used, model predictive control and linear quadratic regulator.

Model Predictive Control

Using a model predictive controller, a finite time-horizon is computed. For each time step leading up to the horizon, the optimal control input for the current time step is computed (via linearizing the model at the current time step), actuated, and then reoptimized to the new time-horizon until the end of the simulation. The main advantage of this control strategy over PID is that the optimization of the control input required for the current time step takes into consideration the future time steps. Furthermore, this enables the controller to anticipate future events and effect appropriate control measures. This control strategy is particularly useful if the vehicle is to automatically detect and avoid obstacles.

Linear Quadratic Regulator

A linear quadratic regulator (LQR) is also a good candidate for a control strategy in such an optimal control problem. Using an LQR involves computing the optimal control inputs that minimize a specified quadratic cost function. In this specific problem, the cost function can be the amount of fuel used that must be minimized. The difference between this and model predictive control is that an LQR optimizes over the entire time horizon of the simulation, while an MPC uses a receding horizon. An advantage of LQR over MPC lies in the lower amount of computational power required.

See Also

6DOF (Euler Angles) | Direction Cosine Matrix to Rotation Angles | Rotation Angles to Direction Cosine Matrix | Three-axis Inertial Measurement Unit

Mars Helicopter Simulink-Based System Level Design

This example shows how to use Simscape™ Electrical™ and Aerospace Blockset™ to model a helicopter with coaxial rotors suitable to fly on Mars. This helicopter takes inspiration from `Ingenuity`, the robotic helicopter developed by NASA, which accomplished the first powered flight on another planet.

Model Overview

The helicopter model comprises a solar panel, a battery pack, a heater, a motor and drive, two gearboxes, two contra-rotating coaxial rotors, and a 1D mechanical model of the gravity, drag, mass, and ground contact forces.

The parameters for the elements in the model are saved in `marsHelicopterSimulinkbasedSystemData.mat`.

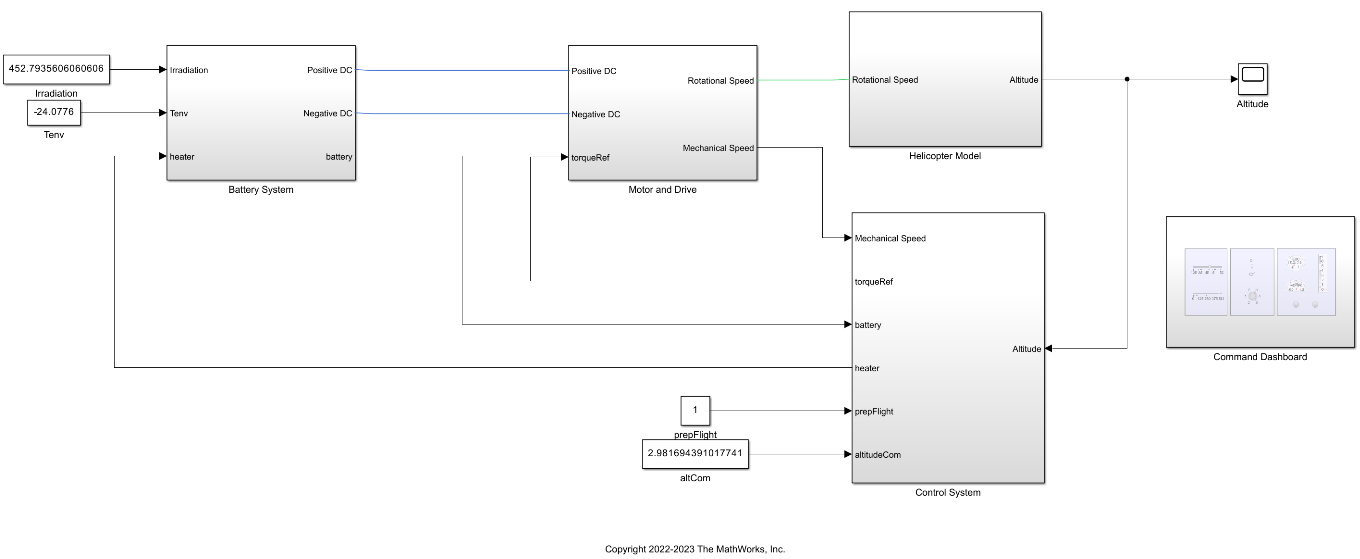
To open the model, use the following command:

```
model = 'marsHelicopterSimulinkbasedSystem';
open_system(model);
```

The `Preload` function loads the MAT-file and adds the struct 'ingenuityParams' to the workspace. The struct includes the environment and electrical and mechanical parameters of the model. To study the effects of different parameters on the system performance, you can change the values with the code in the `marsHelicopterSimulinkbasedSystemParams.m` file:

```
open('marsHelicopterSimulinkbasedSystemParams.m')
```

The basic overview of the Simulink® model is as shown:

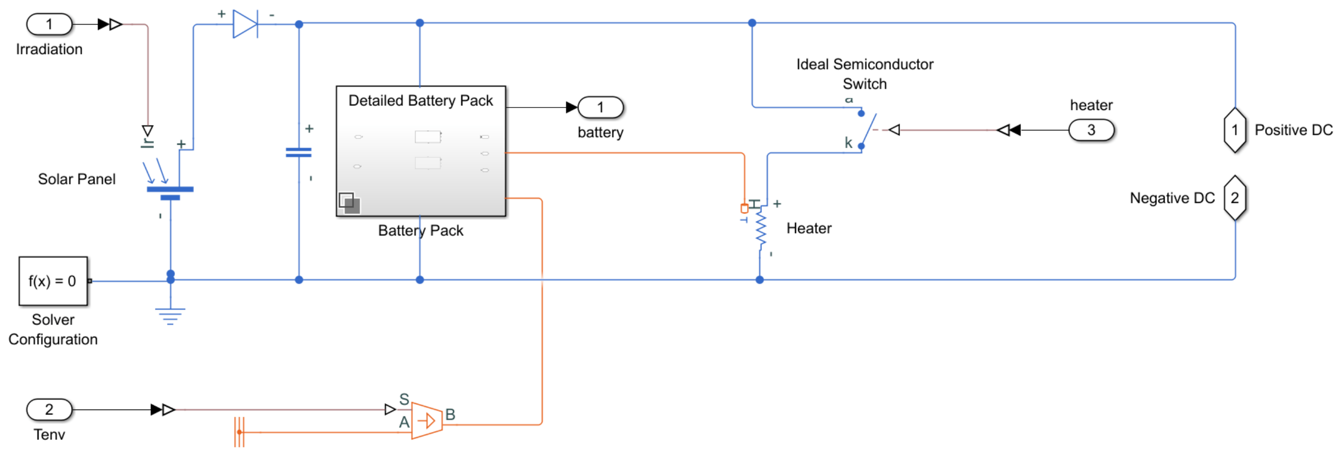


The Battery System involves the solar panel and the batteries required to power the helicopter. The Motor and Drive and the Helicopter Model subsystems model the coaxial helicopter system, which produces the required thrust based on the reference torque input. The Control System, which consists of PID controllers, computes the reference torque based on the altitude command. To control the helicopter altitude interactively, use the blocks in the Command Dashboard subsystem.

Battery System

The Battery System consists of the Solar Panel, the Battery Pack, Heater, and an Ideal Semicondutor Switch.

- The inputs to the system are the incident irradiance to the solar cell and the physical signal to the gate terminal of the ideal semiconductor switch.
- The system outputs a bus signal, which consists of the state of charge (SOC) and temperature of the battery pack. This output signal determines when the Control System gets activated.



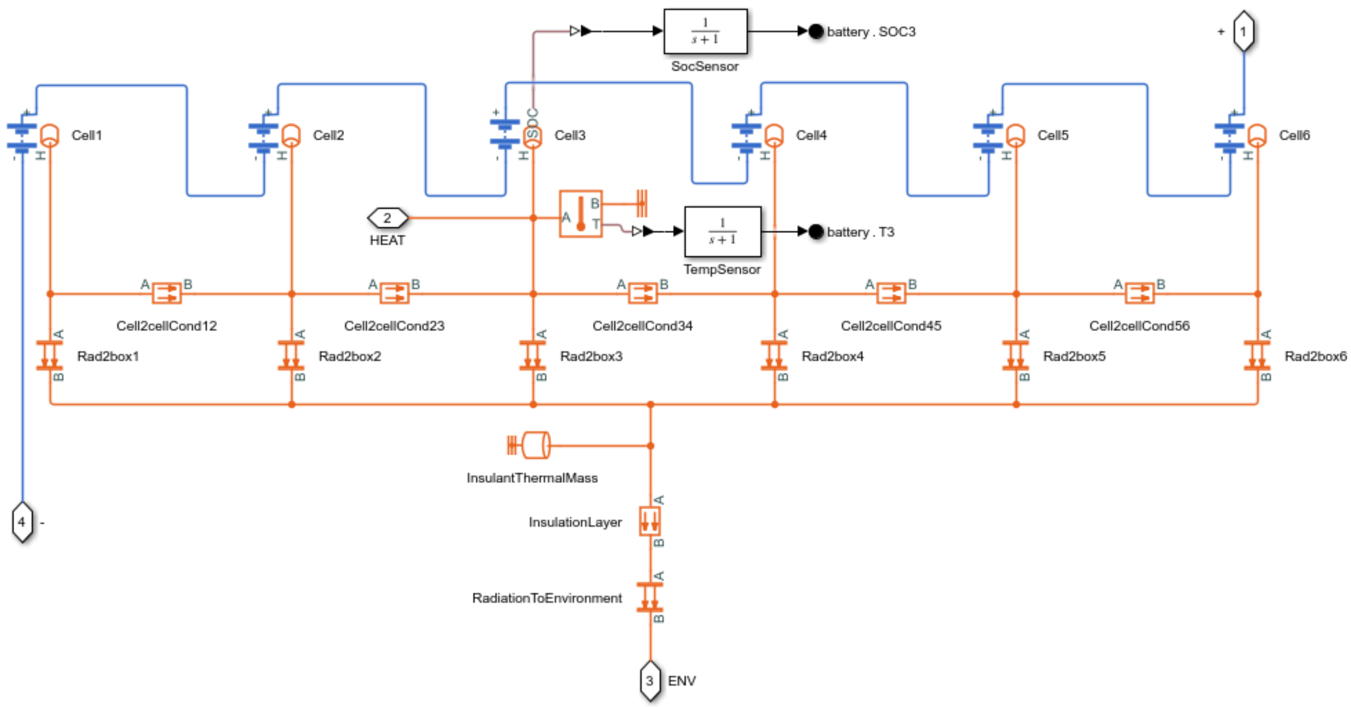
The Battery Pack is a Variant subsystem with two variant options, 'Detailed Battery Pack' and the 'Simplified Battery Pack'.

Select the appropriate variant based on use case. In the Simulink toolstrip, select Variant Subsystem > Open and click a variant from the list. Alternatively, you can use the function `'marsHelicopterSimulinkbasedSystemSetBattery'` to select the appropriate variant as shown here:

```
batteryVariant = 'Detailed'; % batteryVariant = 'Simplified';
marsHelicopterSimulinkbasedSystemSetBattery(model, batteryVariant);
```

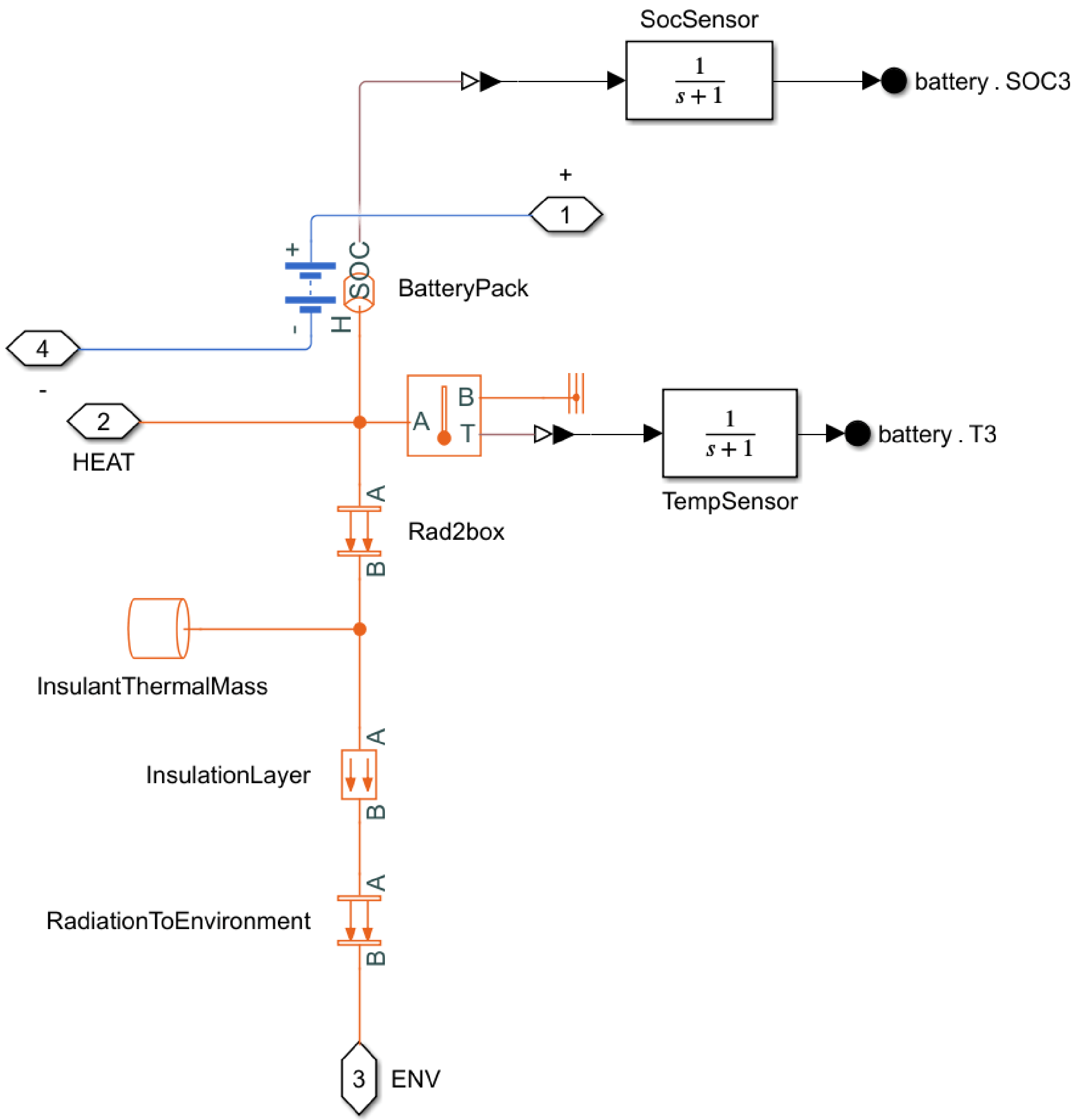
Detailed Battery Pack

The battery pack contains six lithium-ion cells connected in series. Only the central cell is instrumented. The Heater is attached to the central cell. The cells transfer heat to each other and to the insulating box, which dissipates heat to the martian environment.



Simplified Battery Pack

The battery pack in this case contains a single lithium-ion cell.



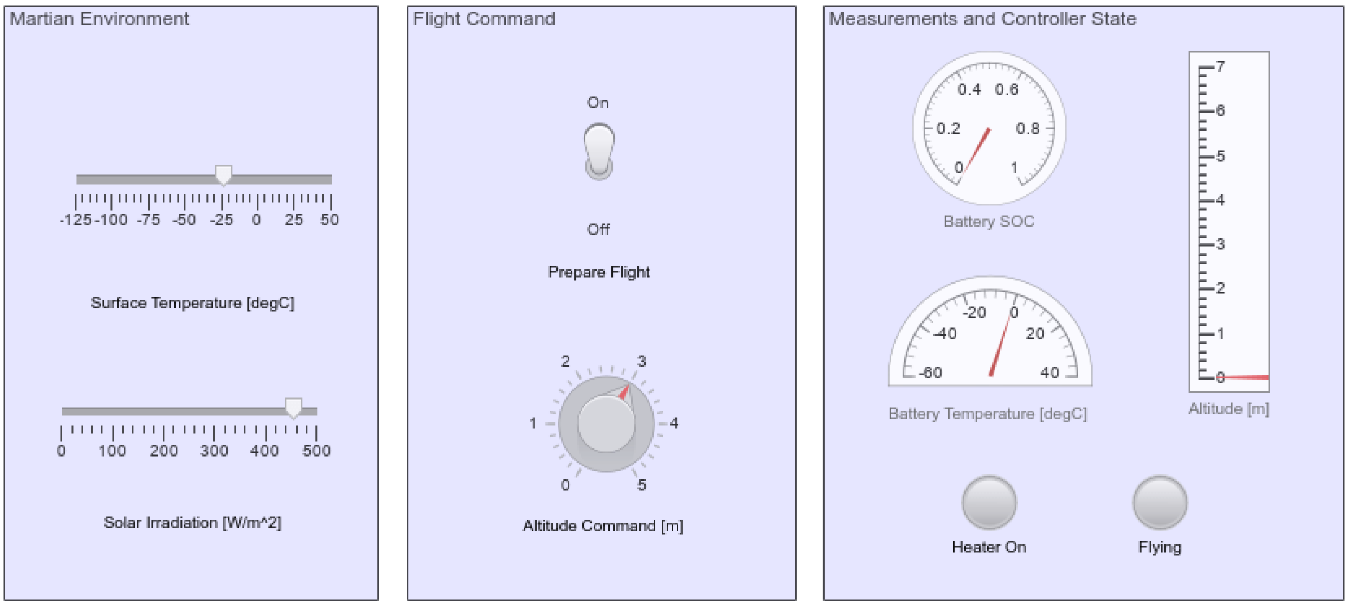
By default, the 'Detailed Battery Pack' variant is selected.

To correctly size the battery pack, consider the nominal voltage of the motor drive and the required nominal power, the nominal flight duration, the payload weight, and the battery energy to weight ratio. Additionally, the solar panel size and the average sun irradiation, together with the heat required to sustain the minimum temperature during the cold martian nights, determine the minimum and maximum energy that you can store during each daily cycle.

Command Dashboard

To control the helicopter interactively during the simulation, use the Sliders, Switches, and Knob blocks in the Command Dashboard subsystem.

As the model solves the equations much faster than real time, simulation pacing is enabled to slow down the animations to only 30 times faster than real time.

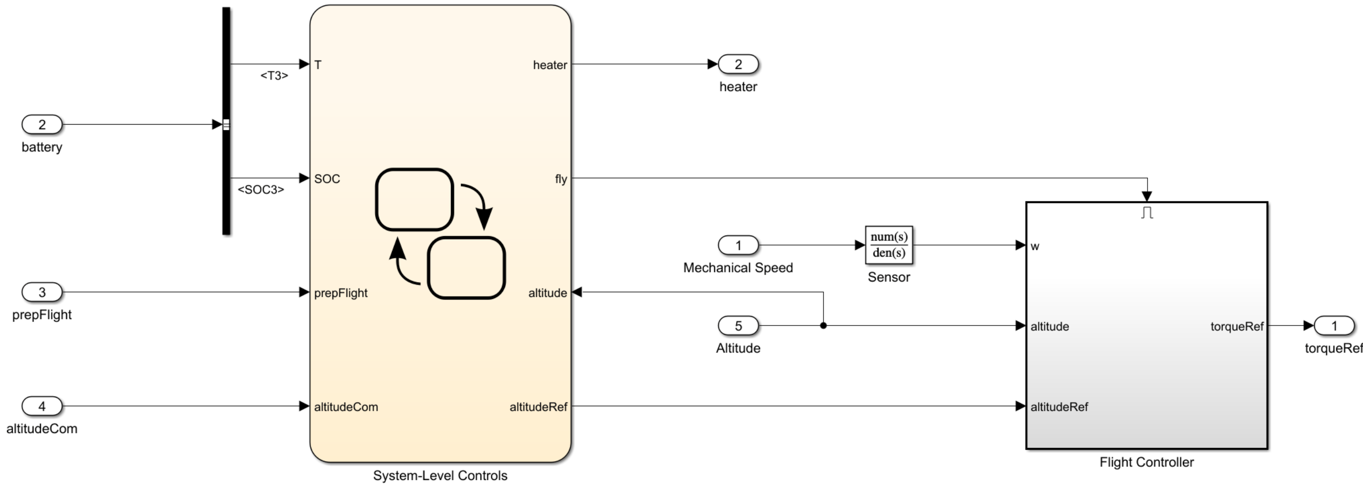


Control System

The helicopter must survive the cold martian nights, which can reach temperatures as low as -125 degC (-193 degF). The electronics and batteries cannot operate at such temperatures, so a heater (part of the Battery system) keeps the temperatures in an acceptable range.

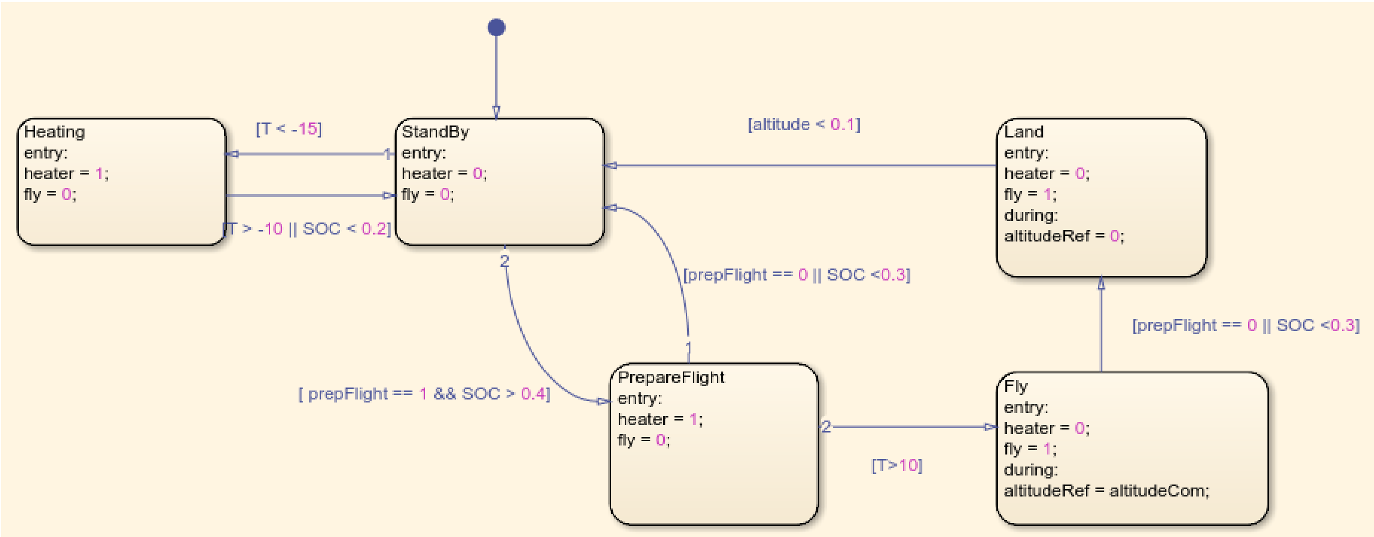
The rotors consume high amounts of power during flight, but the batteries can only provide such power above certain temperatures. As a result, before taking off, we must use the heater to reach these temperatures.

The inputs to the control system include the battery bus signal (SOC and temperature) and the mechanical speed of the Motor and Drive subsystem.

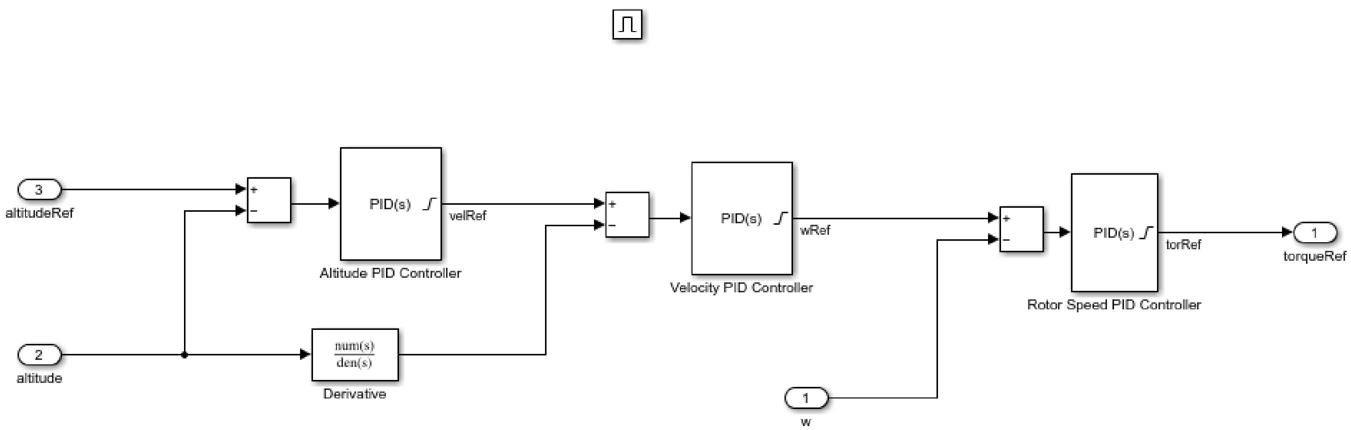


To implement the logic to turn on or off the triggered 'Flight Controller' subsystem, use the state chart 'System-Level Controls'. Based on the battery conditions and the altitude command input, the state chart also provides the reference altitude. The necessary conditions for flight ('Fly' state) in this scenario are:

- Temperature (T) must be above 10 degC (50 degF).
- SOC must be above 0.3.

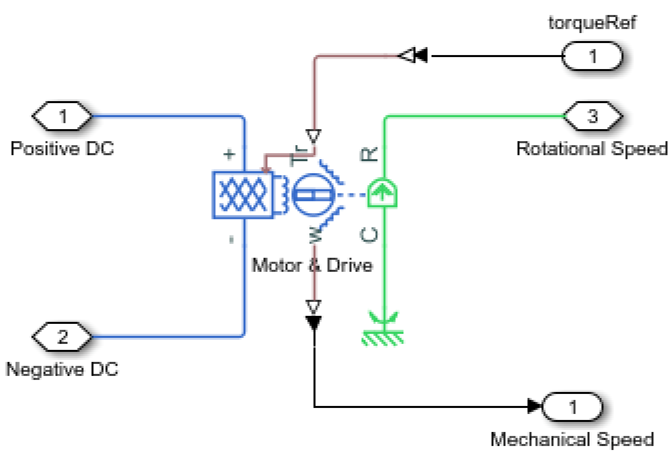


The reference altitude, measured altitude, and the mechanical speed form inputs to the Flight Controller subsystem. Once the helicopter is ready to fly, the Flight Controller subsystem is enabled to control the altitude. The Flight Controller subsystem comprises three PIDs that control the rotor speed, the climb rate, and the altitude. The output is the reference torque for the Motor and Drive subsystem.

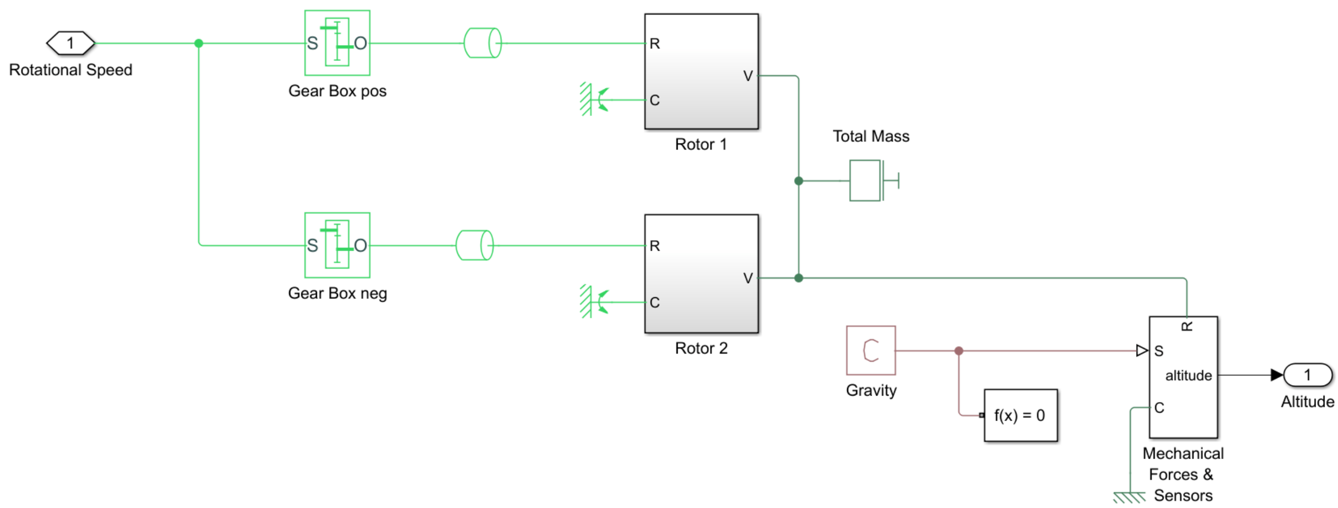


Motor and Drive

The reference torque as computed by the Flight Controller is input to the Motor & Drive block. This block then provides the torque input to the two rotors of the Helicopter system via respective gear boxes.



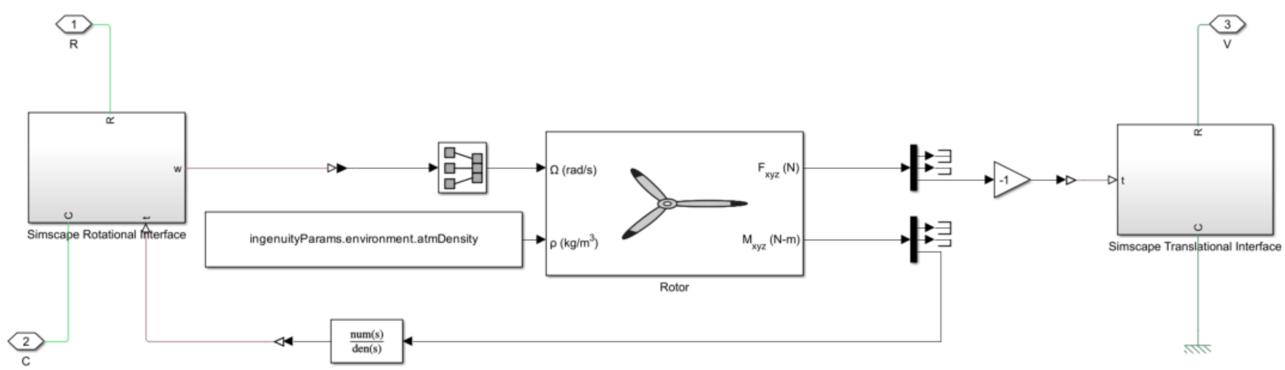
Helicopter Model



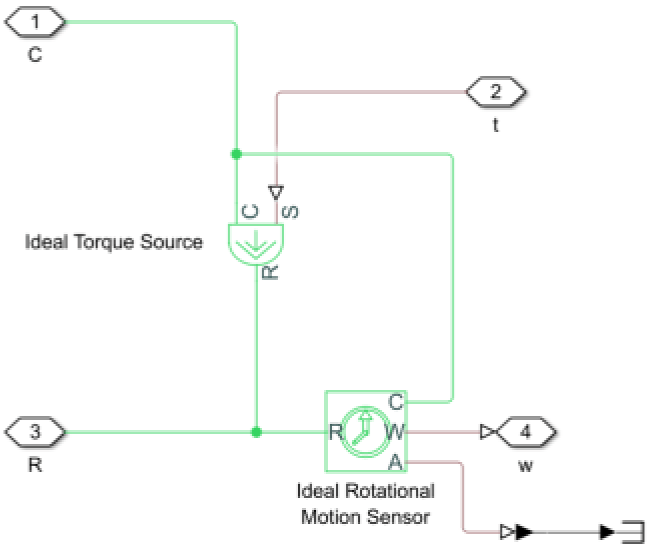
The positive and negative gear boxes signify that in the coaxial system, the two rotors rotate at the same speed but in opposite directions, thereby balancing the torque. In this specific scenario, blade pitch control is not considered. The Mechanical forces & Sensors subsystem represent the body and landing gear of the helicopter (mass and contact forces).

The Rotor block is an Aerospace Blockset block, therefore interface subsystem blocks are used to convert between the physical (Simscape) and Simulink signals. The interfaces are used at the input (rotational) and output (translational) end of the rotor system. The required inputs to the Rotor block are the rotational velocity and the density. The rotor, in this scenario, performs pure vertical motion, therefore there exists no significance in including flap dynamics.

As a result, the modeling approach in the Rotor block is set to 'Without flap effects'. In this case, the required parameters directly input to the block are the thrust, torque coefficients, and the blade radius.

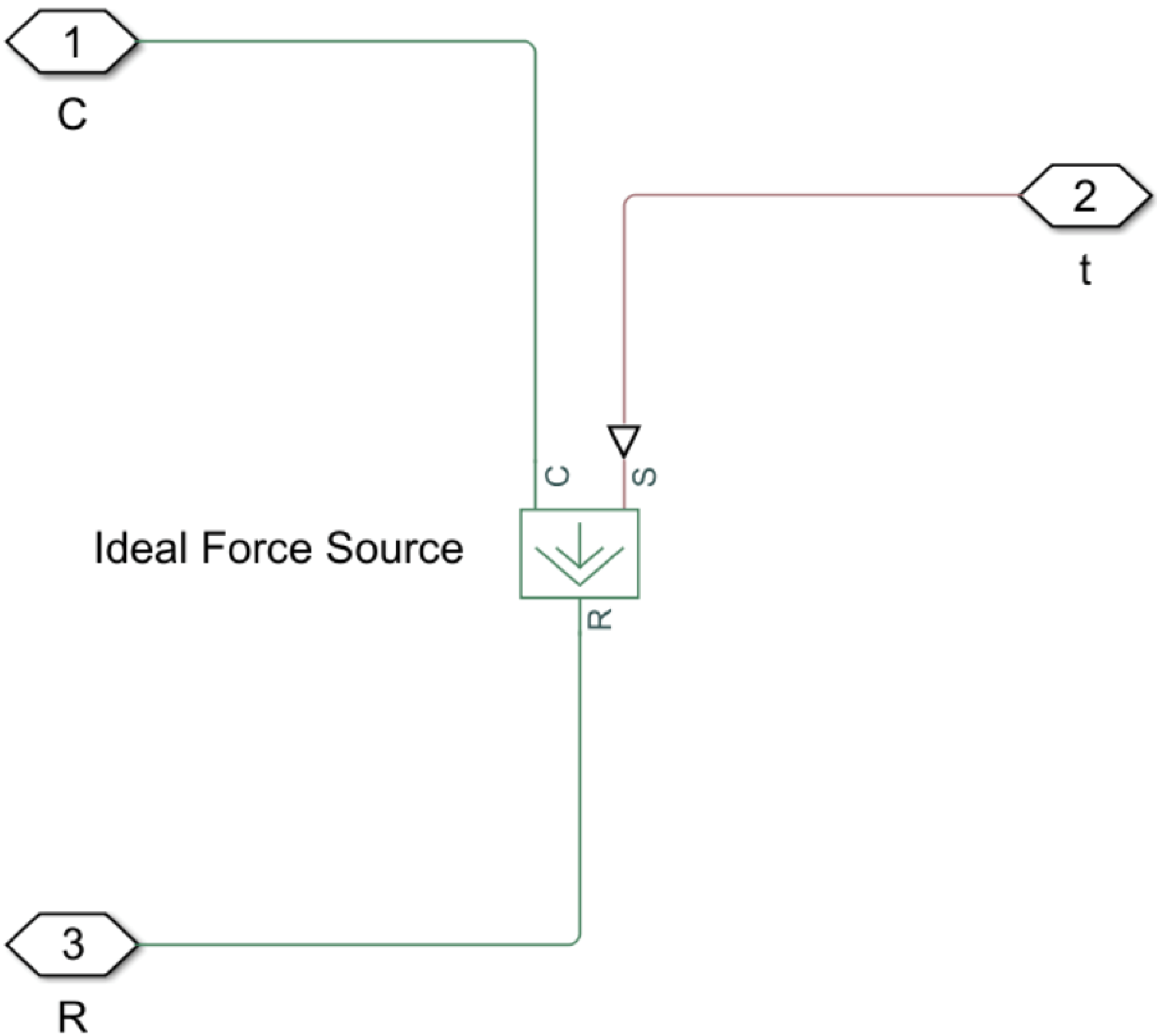


The Simscape Rotational Interface, using an Ideal Torque Source and Ideal Rotational Motion Sensor, computes the required rotational velocity from the input torque.



The outputs from the Rotor block are the forces and moments in all three dimensions. In this scenario, sole interest is in the force in the z-direction and moment about z-axis. The Rotor block uses a z-downward positive convention. To compensate for the difference in convention, a negative gain is added.

The Simscape Translational Interface using an Ideal Force Source computes the equivalent translational velocity from the input force, which is used in the control system for the computation of altitude.



Results

Simulate the Model

To simulate the model with 'Detailed' Battery Pack enabled:

```
if ~exist('outMarsHelicopterSimulinkbasedSystem', 'var') || ...
        ~strcmp(get_param([model, '/Battery System/Battery Pack'], 'LabelModeActiveChoice'), 'De
    set_param([model, '/Battery System/Battery Pack'], 'LabelModeActiveChoice', 'Detailed Batter
    set_param(model, 'EnablePacing', 'off');
    out = sim(model);
    set_param(model, 'EnablePacing', 'on');
```

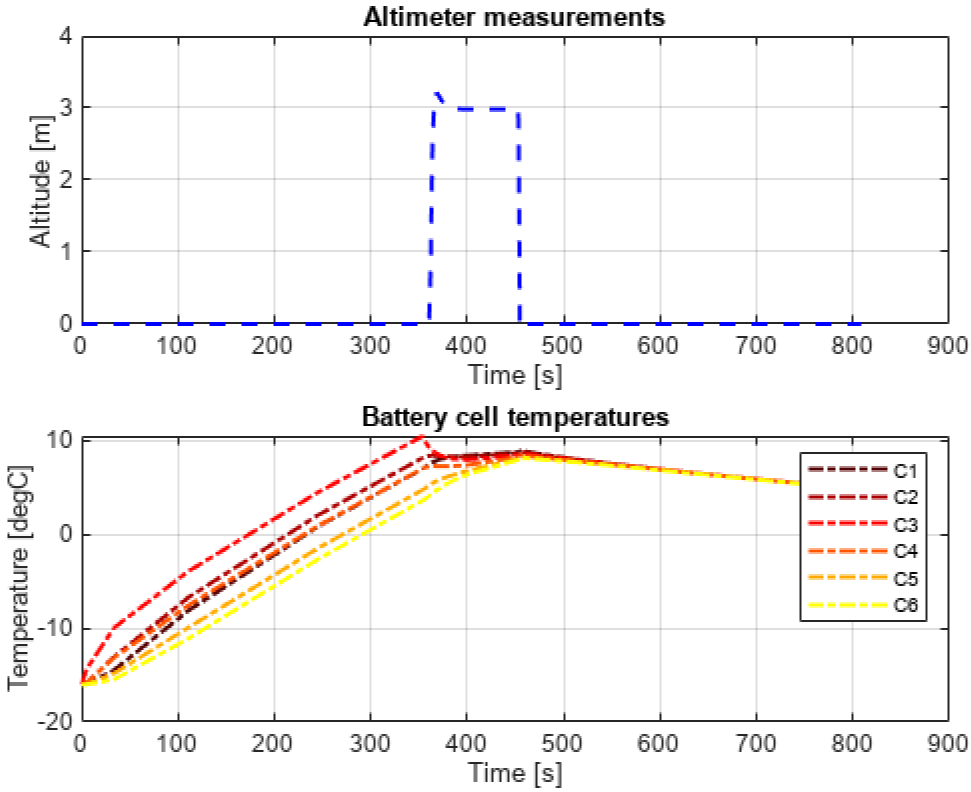
```
else
    out = outMarsHelicopterSimulinkbasedSystem;
end
```

Plot Altitude and Battery Cell Temperatures

The 'Altitude' scope displays the variation in altitude over the entire duration.

The altitude and battery cell temperature variation are plotted below:

```
simlog = out.simlog_marsHelicopterSimulinkbasedSystem;
tout = simlog.Helicopter_Model.Mechanical_Forces_Sensors.Ideal_Translational_Motion_Sensor.x.seri
altitude = simlog.Helicopter_Model.Mechanical_Forces_Sensors.Ideal_Translational_Motion_Sensor.x
fig_plot1 = figure('Name', 'marsHelicopterSimulinkbasedSystemPlot1');
figure(fig_plot1)
clf(fig_plot1)
subplot(2,1,1)
plot(tout, altitude,'b--', 'LineWidth',1.5)
grid on
xlabel('Time [s]')
ylabel('Altitude [m]')
title('Altimeter measurements')

tout = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell1.cell_temperature.series.tim
T1 = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell1.cell_temperature.series.value
T2 = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell2.cell_temperature.series.value
T3 = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell3.cell_temperature.series.value
T4 = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell4.cell_temperature.series.value
T5 = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell5.cell_temperature.series.value
T6 = simlog.Battery_System.Battery_Pack.Detailed_Battery_Pack.Cell6.cell_temperature.series.value

subplot(2,1,2)
colorOrder = hot(10);
plot(tout, [T1,T2,T3,T4,T5,T6], '-.','LineWidth',1.5);
grid on
set(gca, 'ColorOrder', colorOrder)

xlabel('Time [s]');
ylabel('Temperature [degC]');
title('Battery cell temperatures');
legend({'C1', 'C2', 'C3', 'C4', 'C5', 'C6'});
```




```
% Clean-up
clear out colorOrder simlog tout altitude T1 T2 T3 T4 T5 T6
```

As can be observed from the results, the helicopter moves to the 'Fly' state when the battery temperature reaches the required value of 10 degC (50 degF) and moves towards the reference altitude of 3m. The 'Land' state is initiated when the battery SOC falls below the value of 0.3.

Plot Flight Duration Against Number of Battery Cells

The total number of batteries directly influences the maximum possible duration of the flight. To provide more energy storage, you can add more batteries. However, doing so also increases the weight of the vehicle. Eventually, the motor will be unable to provide enough power to lift the extra weight.

The following task studies the effect of number of additional batteries on the flight duration.

1. Setup

Save the original model parameters to restore at the end of the task:

```
ingenuityParamsOriginal = ingenuityParams;
T_marsOriginal = str2double(get_param([model, '/Tenv'], 'Value'));
Irr_marsOriginal = str2double(get_param([model, '/Irradiation'], 'Value'));
missionAltitudeOriginal = str2double(get_param([model, '/altCom'], 'Value'));
```

For this simulation, adjust the environment parameters to a desired value:

```
T_mars = -10;     % Mars surface temperature, in degC
Irr_mars = 0;  % Mars incident solar irradiation, in W/m^2
set_param([model, '/Tenv'], 'Value', num2str(T_mars));
set_param([model, '/Irradiation'], 'Value', num2str(Irr_mars));
```

Input the initial battery temperature and state of charge:

```
ingenuityParams.battery.cell.T_init = -10; % Battery initial temperature, in d
ingenuityParams.battery.cell.SOC_init = 0.97;
```

Choose the range of number of parellel strings of 6 cells to consider (numStringsMin:1:numStringsMax) and select the Simplified Battery Pack variant:

```
numStringsMin = 1;
numStringsMax = 6;
set_param([model, '/Battery System/Battery Pack'], 'LabelModeActiveChoice', 'Simplified Battery I
```

Adjust the simulation stop time and disable simulation pacing and fast restart to increase speed:

```
set_param(model, 'StopTime', '2000');
set_param(model, 'EnablePacing', 'off');
```

Input the altitude command and prepare flight at the start of the simulation:

```
missionAltitude = 5; % Target altitude, in m
set_param([model, '/altCom'], 'Value', num2str(missionAltitude));
set_param([model, '/prepFlight'], 'Value', '1');
```

2. Run Simulations

`Simulink.SimulinkInput` object is used to specify the varying parameters for the model.

```
numStringsVec = numStringsMin:1:numStringsMax;

simin(1:length(numStringsVec)) = Simulink.SimulationInput(model);

for i = 1:length(numStringsVec)
    thisNumStrings = numStringsVec(i);
    simin(i) = simin(i).setVariable('ingenuityParams.battery.numParallelStrings',thisNumStrings)
    simin(i) = simin(i).setVariable('ingenuityParams.battery.numCells',thisNumStrings*ingenuityPa
end

simout = sim(simin,'ShowProgress','off');
```

To simulate the model multiple times using varying parameters, `parsim` function may be used as shown:

```
simout = parsim(simin,'UseFastRestart','on','ShowProgress','off');
```

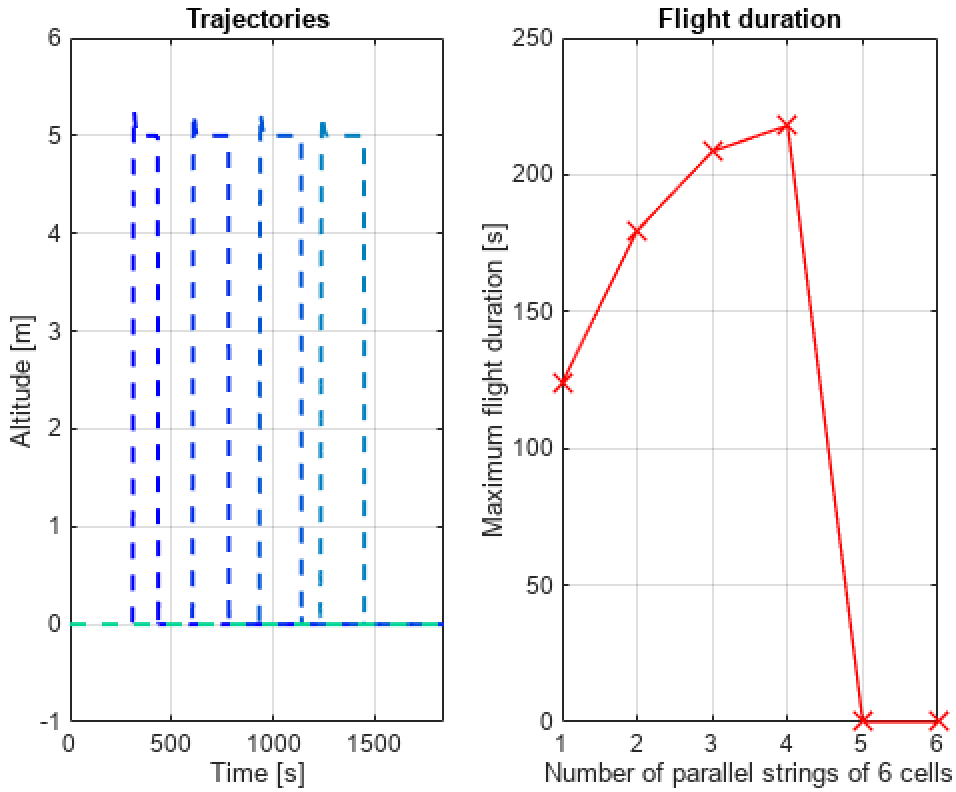
Reset model to original settings:

```
set_param(model, 'StopTime', '15*60');
set_param(model, 'EnablePacing', 'on');
ingenuityParams = ingenuityParamsOriginal;
set_param([model, '/Tenv'], 'Value', num2str(T_marsOriginal));
set_param([model, '/Irradiation'], 'Value', num2str(Irr_marsOriginal));
set_param([model, '/altCom'], 'Value', num2str(missionAltitudeOriginal));
set_param([model, '/Battery System/Battery Pack'], 'LabelModeActiveChoice', 'Detailed Battery Pa
```

3. Analyze Results

Plot each trajectory and flight duration:

```
if ~exist('fig_plot2', 'var') || ...
        ~isgraphics(fig_plot2, 'figure')
    fig_plot2 = figure('Name', 'marsHelicopterSimulinkbasedSystemPlot2');
end
figure(fig_plot2);
clf(fig_plot2);

subplot(1,2,1);
flightDurationVec = nan(size(numStringsVec));
colorOrder = winter(length(numStringsVec)+1);
for i = 1:length(numStringsVec)
    thisAltitude = simout(i).simlog_marsHelicopterSimulinkbasedSystem.Helicopter_Model.Mechanica
    thisTimeVec = simout(i).simlog_marsHelicopterSimulinkbasedSystem.Helicopter_Model.Mechanical_
    % Compute flight duration
    timeStartFly = thisTimeVec(find(thisAltitude > missionAltitude/100, 1, 'first'));
    if isempty(timeStartFly) % Could not take off
        flightDurationVec(i) = 0;
    else
        timeEndFly = thisTimeVec(find( (thisAltitude < missionAltitude/100) & (thisTimeVec > time
        flightDurationVec(i) = timeEndFly - timeStartFly;
    end
    % Plot trajectory
    plot(thisTimeVec, thisAltitude, "--", "LineWidth", 1.5);
    hold on
end
set(gca, 'ColorOrder', colorOrder)
grid on
hold off
xlabel('Time [s]')
ylabel('Altitude [m]')
xlim([0, thisTimeVec(end)])
title('Trajectories')

% Plot flight duration vs number of cells
subplot(1,2,2)
plot(numStringsVec, flightDurationVec, 'rx-', "LineWidth", 1.0, 'MarkerSize', 10);
xlim([numStringsVec(1), numStringsVec(end)])
xticks(numStringsVec)
grid on
xlabel(['Number of parallel strings of ', num2str(ingenuityParams.battery.numSeriesCells), ' cel
ylabel('Maximum flight duration [s]')
title('Flight duration')
```




```
%Clean-up
clear colorOrder flightDurationVec i ingenuityParamsOriginal Irr_mars missionAltitude model numS
clear numStringsMin numStringsVec simout T_mars thisAltitude thisNumStrings thisTimeVec timeEndF
clear Irr_marsOriginal missionAltitudeOriginal T_marsOriginal
```

As seen in these plots, increasing the number of batteries increases the vehicle weight such that the motor can no longer provide sufficient power to lift the vehicle.

References

NASA. Ingenuity Mars Helicopter, Landing Press Kit. Available online: Ingenuity Mars Helicopter

See Also

Blocks
Rotor

Objects
`satelliteScenario`

Related Examples

- "Dashboard"

Collision Avoidance Maneuver for Upcoming Conjunction

This example shows how to identify and avoid an upcoming conjunction. The example use two-line elements (TLEs) to identify the conjunction. It then shows how to use a satellite maneuver with Aerospace Blockset™ to avoid the conjunction. The example uses the Orbit Propagator block to calculate the trajectory after the maneuver.

Disclaimer - Do not use publicly available TLEs for operational conjunction assessment prediction and avoidance. This example is provided for informational purposes only. Satellite operators should contact the 18th Space Defense Squadron for data and analysis to support operational satellites.

Introduction

Due to the large constellations launched in the last few years, satellite conjunctions that require avoidance maneuvers are a common problem in Low Earth Orbit (LEO). In return, satellite operators must respond to conjunction messages received from 18th Space Control Squadron and provide updated ephemerides. NASA addresses this issue in their Best Practices Handbook [1] and the Space Control Squadron in their Spaceflight Safety Handbook for Satellite Operators [2]. Avoidance maneuvers are very small maneuvers, typically along the track of the orbit, to maximize distance from the potential conjunction, as described in [3].

Notes:

- This example uses a Starlink satellite.
- All computations in this example are performed in the inertial frame.

Tools used:

- To find upcoming conjunctions between a pair of objects using their Two-Line Elements, or TLEs, we first use `findConjunctions()` function using the `satellite()` and `states()` functions from the Aerospace Toolbox.
- To find upcoming conjunctions, we use the SOCRATES online tool from the Celestrak website. The probability of the conjunction is computed both before and after the planned avoidance maneuver.
- To compute a maneuver from the desired radial offset and determine the expected tangential miss distance, we use an approximate analytic solution [3]. This is equivalent to the first portion of a Hohmann transfer to a slightly higher orbit, which provides both radial and tangential distance from the original conjunction location.
- To simulate the orbit of the active satellite, we use the Orbit Propagator block. The orbit of the debris is propagated using its TLE. The TLEs must be accurate within about a kilometer for the time frame of interest, which is up to seven days. The reference orbit and the perturbed orbit is used to plot the relative distance between the maneuvering satellite and the conjunction point.

Identify Upcoming Conjunctions

We use an example with a Starlink satellite and CZ-2D DEB debris. The TLE pair for the satellite and debris is saved into a text file.

```
mission.tleFile = 'Conjunction_Starlink1079_CZ2DDeb.txt';
type Conjunction_Starlink1079_CZ2DDeb.txt
```

```
STARLINK-1079
1 44937U 20001Z   22272.52136386 -.00000745  00000+0 -31120-4 0  9997
2 44937  53.0548 210.2777 0001443  83.0631 277.0522 15.06393387150649
CZ-2D DEB
1 43406U 12073D   22272.74034628  .00008296  00000+0  83006-3 0  9993
2 43406  97.9471 196.5364 0090895 225.9886 133.3815 14.88613293478469
```

Load the TLE into a Scenario

Load the elements into a scenario object and print the orbital elements for inspection. Both satellites are set up with the SGP4 propagator. We save the start time of the scenario for later use.

```
mission.sc  = satelliteScenario;
mission.sat = satellite(mission.sc,mission.tleFile); % two satellites
mission.sat(2).MarkerColor = [1 0.059 1];
mission.sat(2).Orbit.LineColor = [1 0.059 1];
mission.startTime = mission.sc.StartTime;
mission.elements(1) = orbitalElements(mission.sat(1));
mission.elements(2) = orbitalElements(mission.sat(2));
```

The satellites orbital elements, from the TLEs, are displayed below. Note satellite 1, the StarLink satellite, has a negative BStar, which is a form of the ballistic coefficient. Negative values can happen due to the averaging process used to generate TLEs. In this case, it means that drag has a very small impact on the location of this satellite over our period of interest, up to seven days from epoch. The debris object has an order of magnitude larger (positive) BStar. The debris object also has an order of magnitude higher eccentricity.

```
mission.elements(1)
```

```
ans = struct with fields:
                      MeanMotion: 0.0628
                    Eccentricity: 1.4430e-04
                     Inclination: 53.0548
    RightAscensionOfAscendingNode: 210.2777
              ArgumentOfPeriapsis: 83.0631
                      MeanAnomaly: 277.0522
                           Period: 5.7356e+03
                            Epoch: 29-Sep-2022 12:30:45
                            BStar: -3.1120e-05
```

```
mission.elements(2)
```

```
ans = struct with fields:
                      MeanMotion: 0.0620
                    Eccentricity: 0.0091
                     Inclination: 97.9471
    RightAscensionOfAscendingNode: 196.5364
              ArgumentOfPeriapsis: 225.9886
                      MeanAnomaly: 133.3815
                           Period: 5.8041e+03
                            Epoch: 29-Sep-2022 17:46:05
                            BStar: 8.3006e-04
```

Find the Conjunctions

Next, find the conjunctions using the `findConjunctions` function. This may take a minute or two depending on the number of close intersections of the orbits within the screening period. The two-step process first uses a coarse propagation with the screening distance to find conjunction windows. A search then finds the closest distance within each window.

```
nDaysDouble = 7;
nDays = days(nDaysDouble);
dScreen = 200000;
dMin = 1000;
[conj.tCA,conj.missDistance,conj.relVelocity] = findConjunctions(mission.sc,...
  mission.sat(1),mission.sat(2),nDays,dScreen,dMin);
disp(table(conj.tCA, conj.missDistance, conj.relVelocity, days(conj.tCA-mission.startTime), ...
  VariableNames=["Conjunction time", "Dist at closest approach (m)", "Velocity difference (m/s)"
```

```
     Conjunction time      Dist at closest approach (m)    Velocity difference (m/s)    Time fr
    ____________________    ____________________________    _________________________    _______

    03-Oct-2022 08:06:49               17.174                         5861.3
```

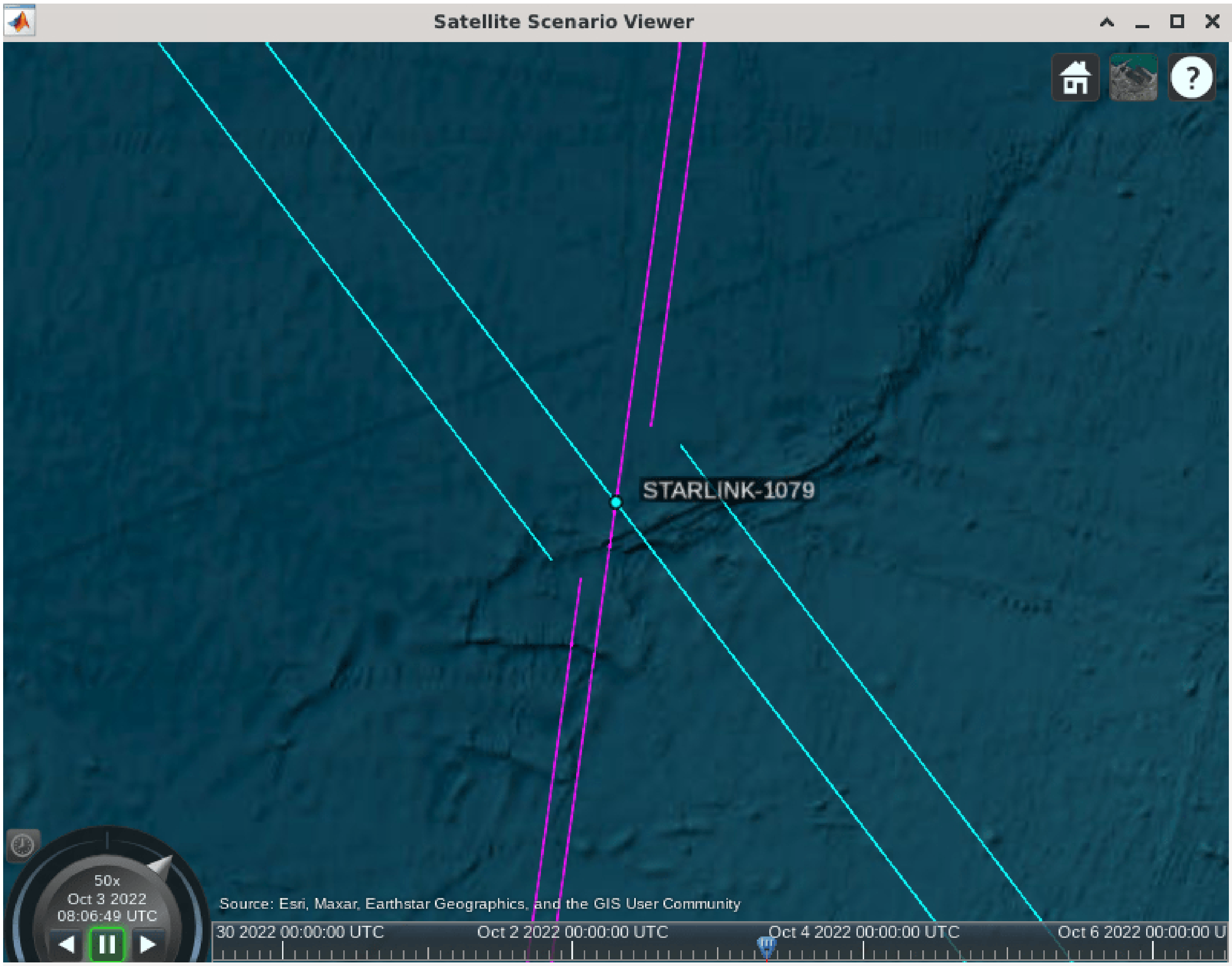
We need the states at the time of closest approach.

```
[conj.X_icrf,conj.V_icrf] = states(mission.sat,conj.tCA(1),CoordinateFrame='inertial'); % m/s
```

Visualize at Time of Closest Approach

View the satellite, STARLINK-1079, and debris, CZ-2D DEB, orbits in the scenario viewer. This zoomed-in view at the time of closest approach shows that the satellite and debris have a probable conjunction.

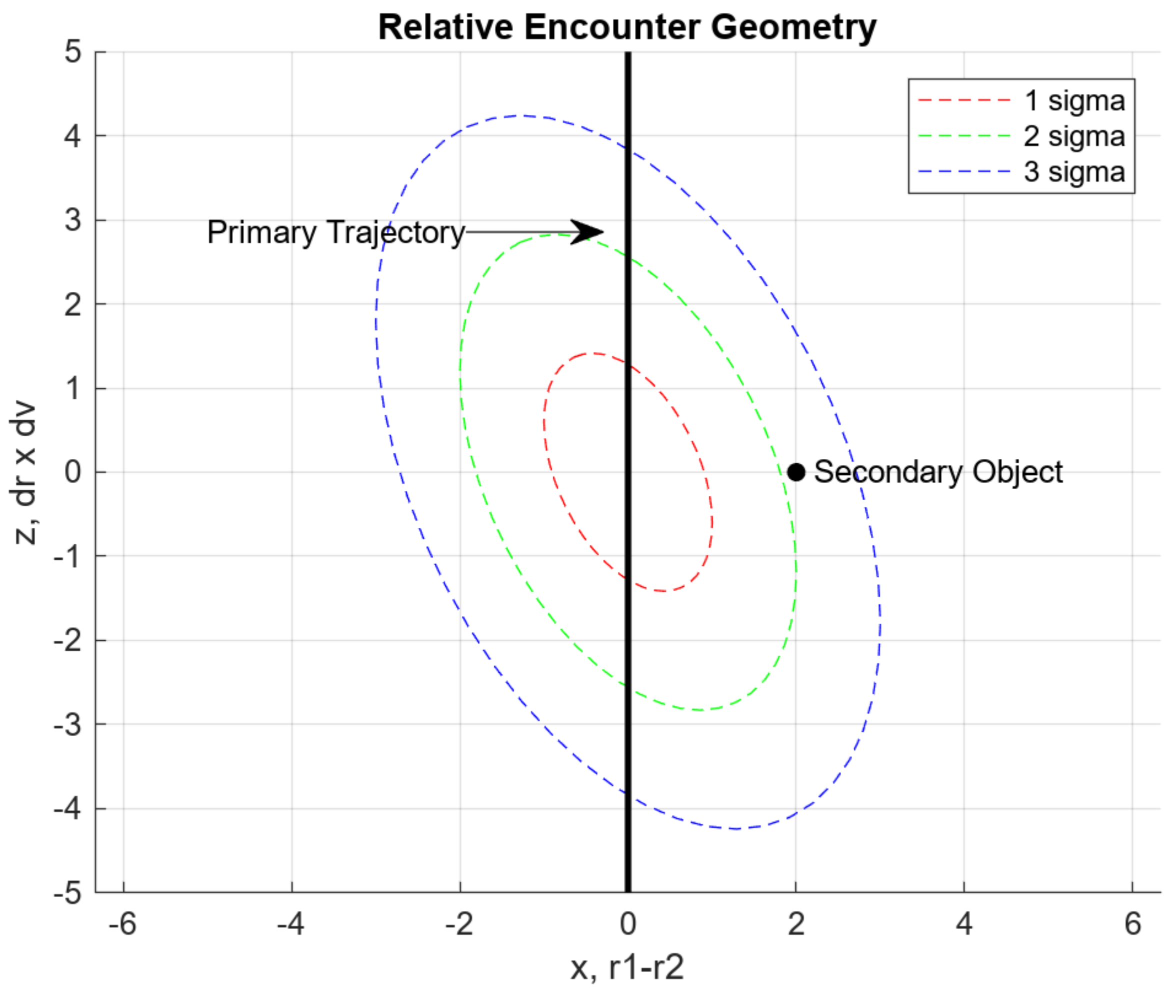
```
viewerMission= satelliteScenarioViewer(mission.sc,"CurrentTime",conj.tCA);
camheight(viewerMission, 9.0e+05);
```



Conjunction Probability

As described in the NASA handbook, the conjunction probability can be computed using integration with `quad2D()`. This requires knowledge or an estimate of the covariance of both objects and their hard-body radii. If the predicted miss distance is within the combined hard-body radii of the satellites, the probability is 1. Otherwise, the combined covariances are integrated over this hard-body sphere. This integration is shown in the plot generated below with the `drawConjunction` function. The miss distance of the secondary object is on the *x*-axis and the primary object's motion is along the *y*-axis. The axes of the encounter frame are computed from the differences in position and velocity and their cross-product, $\Delta\vec{r} = \vec{r_1} - \vec{r_2}$, $\Delta\vec{v} \equiv \vec{y} = \vec{v_1}$-$\vec{v_2}$, $\vec{h} \equiv \vec{z} = \Delta\vec{r} \times \Delta\vec{v}$, $\vec{x} = \vec{y} \times \vec{z}$. The covariance defines the one-sigma ellipse, shown with the red dashed line.

```
cov2x2 = [1 -0.6; -0.6 2]; % combined covariances, [x z]
rAvoid = 0.1;              % avoidance radius
dMiss  = 2;
drawConjunction(cov2x2, rAvoid, dMiss)
```



The amount of probability density falling within the hard-body-radius circle is given by the following integral [3]:

$$P_c = \frac{1}{2\pi\sqrt{\det(C)}}\int_{-R_C}^{R_C}\int_{-\sqrt{R_C^2 - x^2}}^{\sqrt{R_C^2 - x^2}}\exp\left[-\frac{1}{2}\Delta\hat{r}^T C^{-1}\Delta\hat{r}\right]dydx$$

where C is the covariance, $\Delta\hat{r}$ is the miss vector $(\vec{r}_1 - \vec{r}_2)$, and R_c is the combined hard-body radius, $R_1 + R_2$. To compute a probability, we must estimate this combined satellite hard-body radius or avoidance region. A value between 5 and 20 m is reasonable. The International Space Station is an example of a large spacecraft that would need a bigger value. The approximate maximum probability occurs for a particular scaled value of the covariance such that $C_{max} = k^2C$. This computation assumes the probability density is constant over the hard-body sphere, using the value at the sphere's center. The computation is less accurate if the sphere is not small compared to the covariance. The formula is [3]:

$$P_{c,\,max} \approx \frac{R_C^2}{\exp(1)\sqrt{\det(C_{max})}\Delta\hat{r}^T C_{max}^{-1}\Delta\hat{r}} \text{ when } k^2 = \frac{1}{2}\Delta\hat{r}^T C^{-1}\Delta\hat{r}$$

Compute the Inertial Covariances

SOCRATES uses a nominal covariance of 100 m with a bigger value, 300 m, in the tangential direction. Due to orbit dynamics, the covariance naturally elongates in this direction. The other directions, radial and along the orbit normal (cross-track), tend to be equivalent. The ratio between these values is the aspect ratio of the covariance ellipse. This assumed covariance needs to be rotated from the satellites' local frame into the inertial frame before they can be summed for the probability calculation. The local vertical, local horizontal or LVLH frame has axes in the radial, tangential, and orbit normal directions. In the code block below we show how the transformation is computed using the satellite's inertial state at the conjunction. The x axis is along the radial direction, the z axis along the orbit normal and the y axis completes the set.

```
c0_lvlh = diag([100 300 100].^2); % larger errors along-track (m)

% object 1: StarLink satellite
r0 = conj.X_icrf(:,:,1);
v0 = conj.V_icrf(:,:,1);
x = r0/vecnorm( r0 ); % x is radial
h = cross( r0, v0 ); % h is + orbit normal
z = h/vecnorm(h);
y = cross( z, x ); % y completes RHS
conj.A1 = [x'; y'; z'];  % inertial to LVLH matrix
conj.cov1_icrf = conj.A1'*c0_lvlh*conj.A1;
disp(conj.cov1_icrf)
```

```
   1.0e+04 *

    1.7143   -1.2896    1.8817
   -1.2896    3.3285   -3.3975
    1.8817   -3.3975    5.9572
```

With this transformation matrix, we can also compute the components of the miss distance at the conjunction, not just the total miss distance.

```
xMissLVLH = conj.A1*(conj.X_icrf(:,:,2) - conj.X_icrf(:,:,1));
disp(table(vecnorm(xMissLVLH), xMissLVLH(1), xMissLVLH(2), xMissLVLH(3), ...
  VariableNames=["Total Miss Dist (m)", "Radial (m)", "Tangential (m)", "Orbit Normal (m)"]))
```

Total Miss Dist (m)	Radial (m)	Tangential (m)	Orbit Normal (m)
17.174	-0.92671	-15.852	6.5405

In general, there is the most uncertainty in the along-track direction, and much less in the radial direction. This means that the radial direction drives the avoidance maneuver. In this case, the miss distance is less than 1 m in the radial direction. For further calculations, we use the `dcmicrf2lvlh()` function, which computes the transformation matrix from the inertial position and velocity.

```
% object 2: debris
conj.A2 = dcmicrf2lvlh(conj.X_icrf(:,:,2),conj.V_icrf(:,:,2));
conj.cov2_icrf = conj.A2'*c0_lvlh*conj.A2;
```

Collision Probability Variation with Satellite Radius and Covariance Magnitude

Now, we can compute the probability using `collisionProbability2D()` with the states and covariances of the two objects. Any probability over 1e-4 is considered actionable as recommended by the NASA handbook [1]. The maximum probability reported from SOCRATES for this example was 1.363E-02.

Probability Dilution

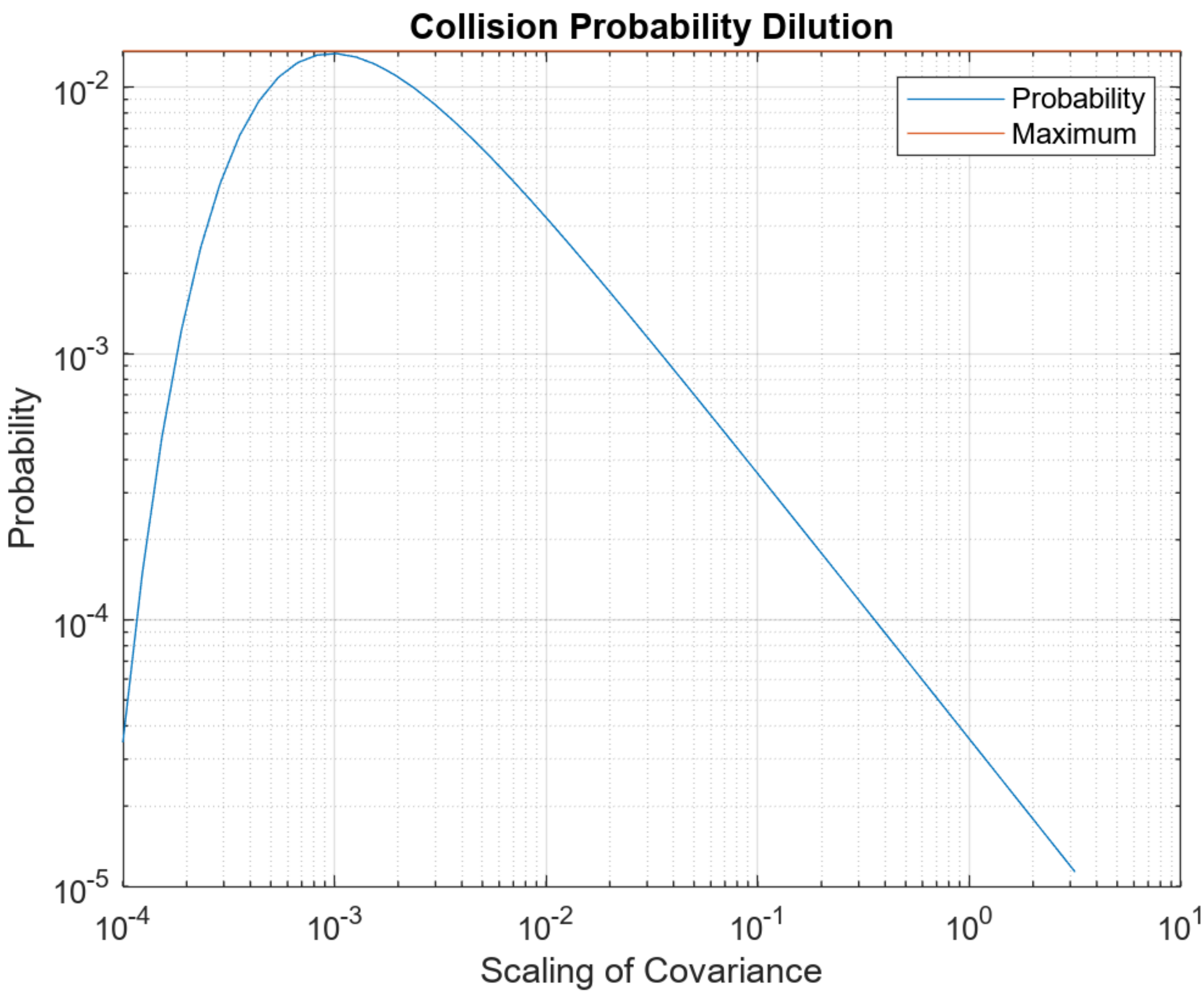
To understand the maximum verses instantaneous probability, let's plot the maximum and the instantaneous probability obtained for a fixed hard-body radius and scaling the covariance. For a very small covariance relative to the miss distance, the probability of collision is small. As the covariance increases, the probability reaches a maximum. Beyond this the probability decreases, known as dilution. This approximate formulation assumes the probability density is constant over the sphere, so we use a small value of the hard-body radius.

```
rSatellite = 2; % use a small value of radius for most accuracy in the comparison
[Pcmax,ksq] = maxCollisionProbability2D(conj.X_icrf(:,:,1),conj.V_icrf(:,:,1),conj.cov1_icrf,...
    conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,rSatellite)
```

```
Pcmax = 0.0137
```

```
ksq = 0.0019
```

```
scale = logspace(-4,0.5);
Pc0sc = zeros(size(scale));
for k = 1:length(scale)
  covC = scale(k)*(conj.cov1_icrf+conj.cov2_icrf);
  Pc0sc(k) = collisionProbability2D(conj.X_icrf(:,:,1),conj.V_icrf(:,:,1),covC,...
    conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),zeros(3),rSatellite);
end
figure
loglog(scale,Pc0sc)
hold on
plot(xlim,Pcmax*[1 1])
grid on
xlabel('Scaling of Covariance')
ylabel('Probability')
title('Collision Probability Dilution')
legend('Probability','Maximum')
```

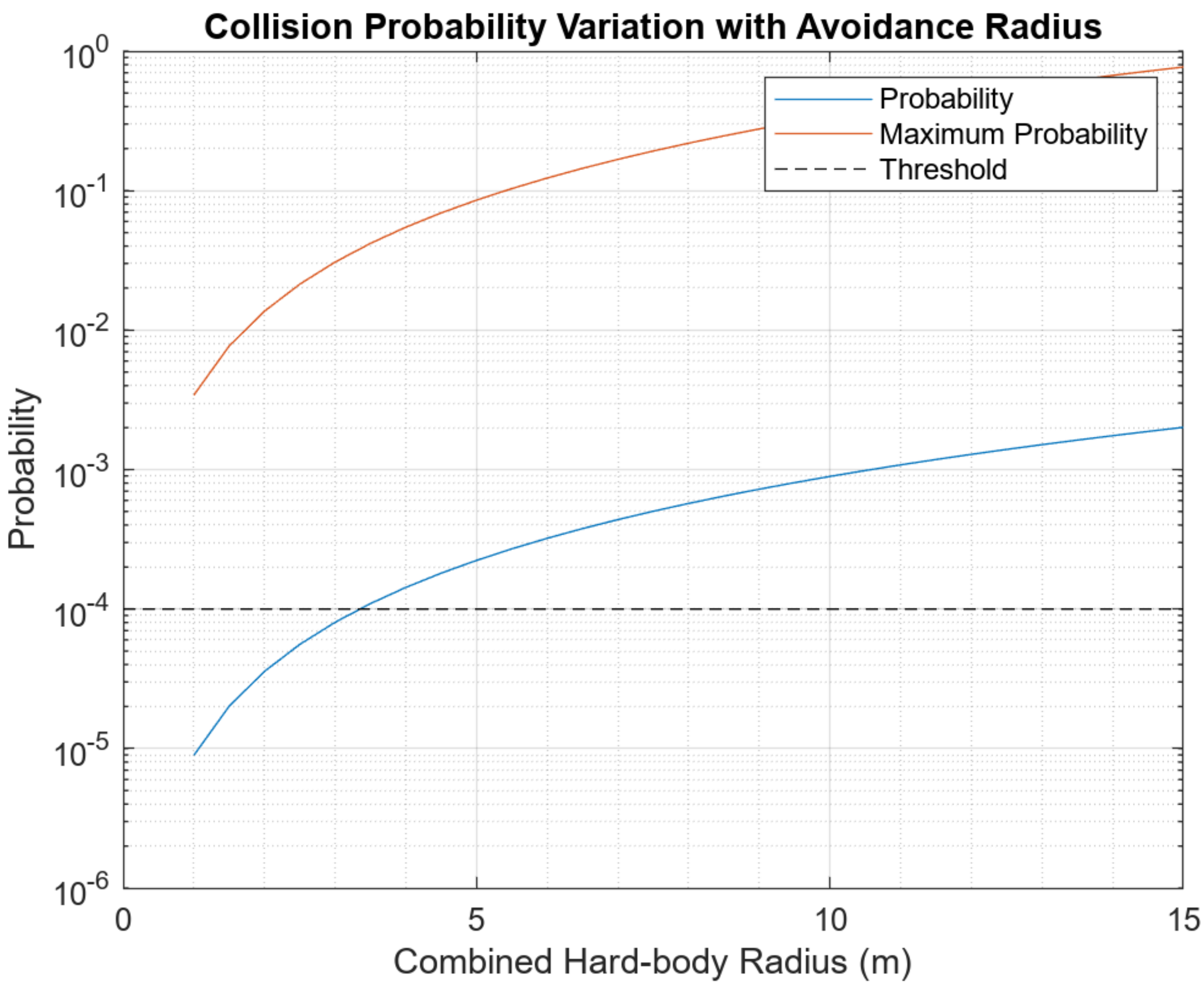


Impact of Avoidance Radius

Our example has a small miss distance of 16 m. If we use a hard-body radius greater than this the probability of collision is 100%. In this code block below, we compute the probability and theoretical maximum for a range of hard-body radii below 15 m.

```
rSatellite = 1:0.5:15; % m
Pc0 = zeros(size(rSatellite));
Pcmax = zeros(size(rSatellite));
for k = 1:length(rSatellite)
  Pc0(k) = collisionProbability2D(conj.X_icrf(:,:,1),conj.V_icrf(:,:,1),conj.cov1_icrf,...
    conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,rSatellite(k));

  Pcmax(k) = maxCollisionProbability2D(conj.X_icrf(:,:,1),conj.V_icrf(:,:,1),conj.cov1_icrf,...
    conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,rSatellite(k));
end
figure('name','Collision Probability')
semilogy(rSatellite,[Pc0;Pcmax])
hold on
plot(xlim,1e-4*[1 1],'k--')
legend('Probability','Maximum Probability','Threshold')
```

```
ylabel('Probability')
xlabel('Combined Hard-body Radius (m)')
title('Collision Probability Variation with Avoidance Radius')
grid on; grid minor
```



Select a Probability for Comparison After the Maneuver

Let's choose a single value of the hard-body radius and resulting collision probability to compare against later, after we execute the maneuver.

```
conj.rSatellite = 8 ;
conj.Pc0 = collisionProbability2D(conj.X_icrf(:,:,1),conj.V_icrf(:,:,1),conj.cov1_icrf,...
    conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,conj.rSatellite);
conj.Pcmax = maxCollisionProbability2D(conj.X_icrf(:,:,1),conj.V_icrf(:,:,1),conj.cov1_icrf,...
    conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,conj.rSatellite);

disp(table(conj.rSatellite, conj.Pc0, conj.Pcmax, ...
  VariableNames=["Hard-body radius (m)","Probability","Maximum Probability"]))
```

Hard-body radius (m)	Probability	Maximum Probability
8	0.00057166	0.21877

Maneuver Planning

Compute the Maneuver

Compute a maneuver to create separation at the conjunction time. This is a simple partial Hohmann maneuver to increase the orbit radius. Every half orbit, the satellite is at the radial maximum distance from the debris object. Due to the slight increase in mean motion, the satellite drifts relative to its original position tangentially, i.e. change phase. To return to the nominal orbit, the delta-V must be repeated in reverse after the conjunction.

The equations from [3] show how the small delta-V relates to the expected difference in tangential and radial distance achieved from the conjunction point. The radial distance is the same every orbit but the tangential distance increases with time. A similar amount of delta-V is required to return to the original orbital station, if required. Note that these equations are approximate and as real orbits have nonzero eccentricity there is always some difference in the actual change of orbit from that predicted.

$\Delta a \approx 2\frac{\Delta V}{V}a$, $\Delta D_R = 2\Delta a$ so that $\Delta V \approx \frac{\Delta D_R V}{4a}$, and $\Delta D_T \approx 3\Delta V \Delta T$

We will specify the target radial offset ΔD_R. We specify the amount of time ahead of the conjunction for the maneuver using half-periods. From this ΔT, we can predict the tangential offset ΔD_T to be achieved.

We compute the average semi-major axis a from the mean motion n obtained from the TLE using this formula where μ is the gravitational constant of the Earth. The mean motion $n = \sqrt{\mu/a^3}$ in the TLE data is in units of degrees per second, so the period is 360 times its inverse. In this code block, we compute the delta-V and predicted tangential distance from the specific radial offset and number of orbit periods. These computations are in km.

```
mvr.delD_R =  1.5                        ;  % radial offset, km
mvr.nP     =  3.5 ; % number of orbit periods, plus 1/2
mvr.v0     = vecnorm(conj.V_icrf(:,:,1)); % m/s
mu         = 3.98600436e5; % gravitational constant, km^3/sec^2
mvr.sma    = (mu/(mission.elements(1).MeanMotion*pi/180)^2)^(1/3); % km
mvr.dVmag  = 0.25*mvr.v0*mvr.delD_R/mvr.sma; % m/s
mvr.deltaT = mvr.nP*mission.elements(1).Period;
mvr.delD_T = 3*mvr.deltaT*mvr.dVmag*1e-3;

disp(table(mvr.delD_R, mvr.dVmag, mvr.deltaT/3600, mvr.delD_T, ...
  VariableNames=["Radial Offset (km)","Delta-V (m/s)","Time Before Conjunction (hours)","Tangent
```

Radial Offset (km)	Delta-V (m/s)	Time Before Conjunction (hours)	Tangential Offset
1.5	0.41087	5.5762	24.744

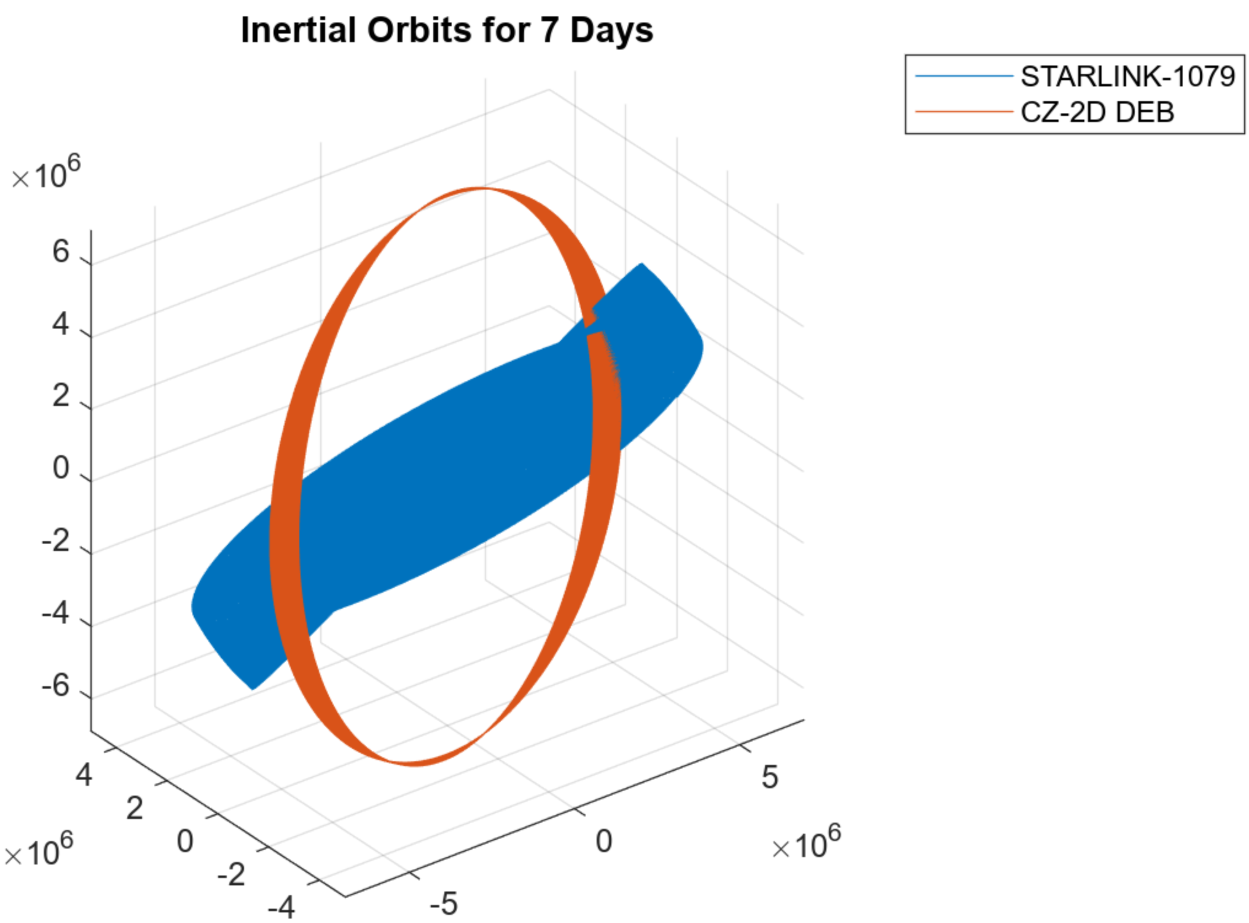
Next, to initialize the numerical integration, compute the states at this maneuver start time using the `states()` function.

```
mvr.mvrTime = conj.tCA - seconds(mvr.deltaT);
[mvr.X0_icrf,mvr.V0_icrf] = states(mission.sat(1),mvr.mvrTime,CoordinateFrame='inertial'); % m/s
```

Obtain the Satellite and Debris Ephemeris up to the Maneuver Time

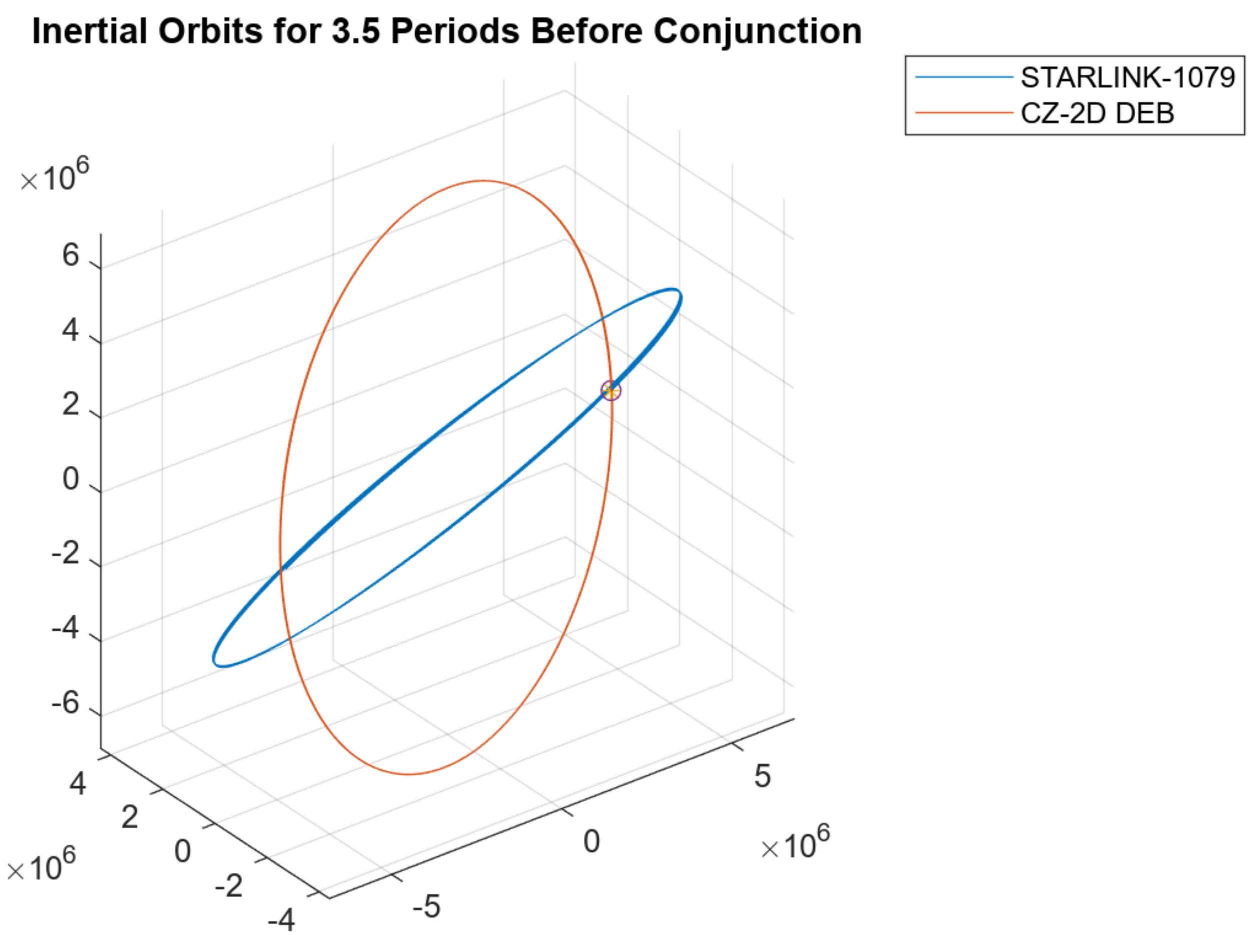
The TLE propagator is SGP4. Here we plot the orbits for the entire seven day conjunction assessment timeframe. Note, the orbits rotate in inertial space due to perturbations from the Earth and other disturbances. The StarLink satellite rotates more than the debris object.

```
mission.sc.StopTime = mission.startTime + nDays;
[pos_all,vel_all] = states(mission.sat,CoordinateFrame='inertial');
figure; plot3(pos_all(1,:,1),pos_all(2,:,1),pos_all(3,:,1))
axis equal
hold on
grid on
plot3(pos_all(1,:,2),pos_all(2,:,2),pos_all(3,:,2))
title(sprintf('Inertial Orbits for %d Days',nDaysDouble))
legend(mission.sat(1).Name,mission.sat(2).Name)
```



Next, plot the nominal orbits between the maneuver time and the detected conjunction.

```
mission.sc.StartTime = mvr.mvrTime;
mission.sc.StopTime = conj.tCA;
[pos_nom,vel_nom,t_out] = states(mission.sat,CoordinateFrame='inertial');
figure; plot3(pos_nom(1,:,1),pos_nom(2,:,1),pos_nom(3,:,1))
axis equal
hold on
grid on
plot3(pos_nom(1,:,2),pos_nom(2,:,2),pos_nom(3,:,2))
plot3(pos_nom(1,end,1),pos_nom(2,end,1),pos_nom(3,end,1),'*')
plot3(pos_nom(1,end,2),pos_nom(2,end,2),pos_nom(3,end,2),'o')
title(sprintf('Inertial Orbits for %g Periods Before Conjunction',...
  seconds(conj.tCA-mvr.mvrTime)/mission.elements(1).Period))
legend(mission.sat(1).Name,mission.sat(2).Name)
```

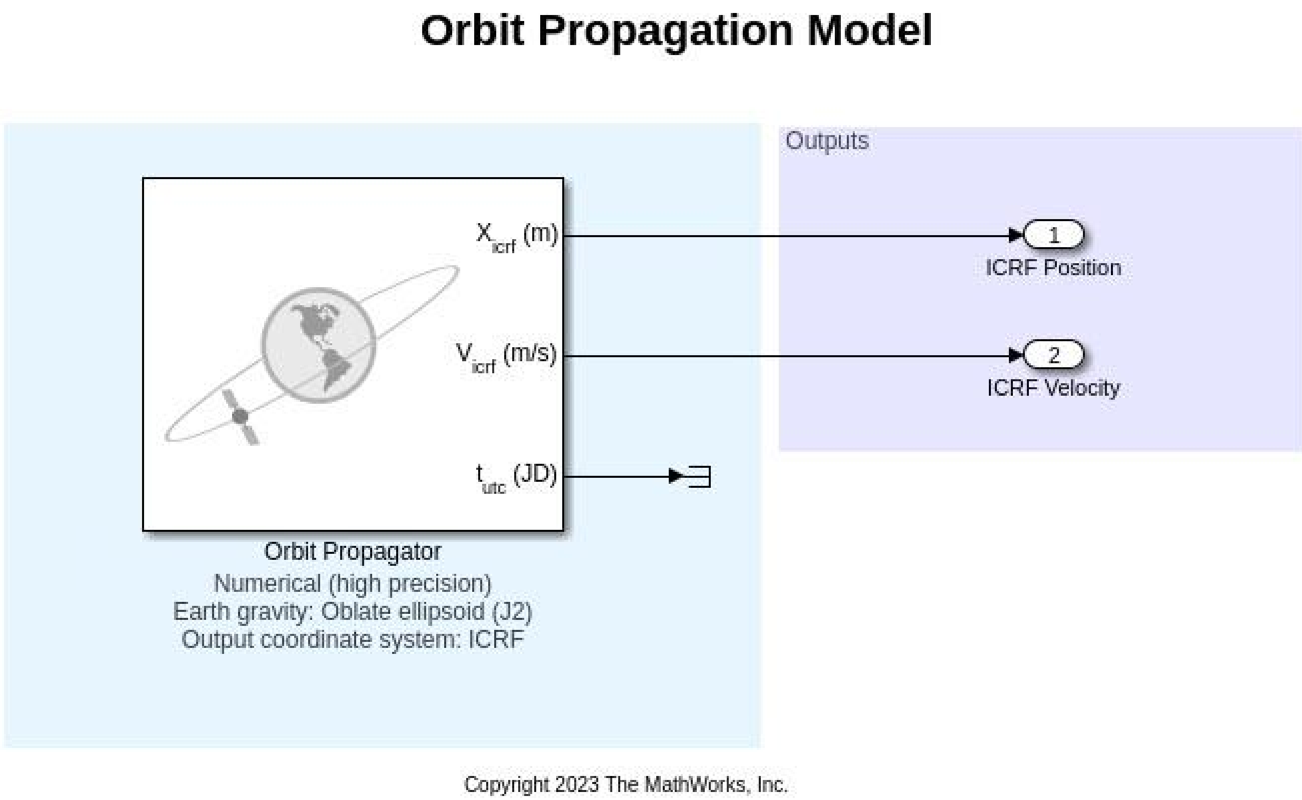


Simulate the Maneuver

Simulate with the Orbit Propagator Block

Load the propagator model with the Orbit Propagator block. Matching SGP4 trajectories requires at least the J2 oblateness term, which defines how the Earth radius is largest at the equator and smallest at the poles. SPG4 uses the zonal only terms of J3 and J4 and no higher order terms. SGP4 include the average effect of drag but with a simple exponential model of the atmosphere. The StarLink satellite has a negative ballistic coefficient, which in this case indicates that drag has very little effect, presumably due to the satellites performing near-continuous drag makeup. First, we open the model.

```
mvr.mdl = "OrbitPropagationModel";
open_system(mvr.mdl);
mvr.blk = mvr.mdl + "/Orbit Propagator";
```



The maneuver delta-V is applied along the velocity unit vector, $\hat{v}$, computed using the velocity magnitude.

```
mvr.vHat = mvr.V0_icrf/vecnorm(mvr.V0_icrf); % velocity direction
```

We simulate both the unperturbed trajectory and the trajectory with the maneuver.

```
X_sim = [mvr.X0_icrf mvr.X0_icrf]';
V_sim = [mvr.V0_icrf mvr.V0_icrf+mvr.dVmag*mvr.vHat]';
```

Set the block and model parameters with `set_param()`.

- We specify the Julian date to significant digits, with a second input to `num2str`.
- Add the delta-V to the state obtained from the TLE elements and passed to `inertialVelocity`.
- Set the stop time on the model using the time between the maneuver and the predicted conjunction. We use a fixed step size to compare the numerically propagated trajectory both with and without the maneuver, i.e. so that the timesteps are the same for both.

```
set_param(mvr.blk, ...
    startDate = num2str(juliandate(mvr.mvrTime),15), ...
    stateFormatNum = "ICRF state vector", ...
    inertialPosition = mat2str(X_sim), ...
    inertialVelocity = mat2str(V_sim), ...
    gravityModel = "Oblate ellipsoid (J2)", ...
    units = "Metric (m/s)", ...
    outportFrame = "ICRF");
set_param(mvr.mdl, ...
    SolverType = "Variable-step", ...
    MaxStep = "20", ...
    RelTol = "1e-12", ...
    AbsTol = "1e-14", ...
    StopTime = string(mvr.deltaT), ...
    SaveOutput = "on", ...
    OutputSaveName = "yout", ...
    SaveFormat = "Dataset", ...
    DatasetSignalFormat = "timetable");
```

Simulate the trajectory after the maneuver by calling the `sim` function on the model.

```
tic
simOutput = sim(mvr.mdl);
toc
```

```
Elapsed time is 5.538485 seconds.
```

Check the Achieved Offset

To compute the new probability and miss distance, get the values from the output timetable.

```
ephPosICRF = simOutput.yout{1}.Values; % data is n×2×3
ephVelICRF = simOutput.yout{2}.Values;

no_mvr.X_icrf = squeeze(ephPosICRF.Data(:,1,:))';
no_mvr.V_icrf = squeeze(ephVelICRF.Data(:,1,:))';
no_mvr.XF_icrf = no_mvr.X_icrf(:,end);
no_mvr.VF_icrf = no_mvr.V_icrf(:,end);
mvr.X_icrf = squeeze(ephPosICRF.Data(:,2,:))';
mvr.V_icrf = squeeze(ephVelICRF.Data(:,2,:))';
mvr.XF_icrf = mvr.X_icrf(:,end);
mvr.VF_icrf = no_mvr.V_icrf(:,end);
```

Let's compare this simulation, before the maneuver, with the previously computed TLE-generated orbit positions at the conjunction, stored in `conj.X_icrf`. The results should be within 1 km, and are in fact less than 100 m.

```
deltaTLE = vecnorm(no_mvr.XF_icrf - conj.X_icrf(:,:,2)) % m
```

```
deltaTLE = 78.0265
```

Now let's compute the new distance achieved with the maneuver. We compute the distance between the endpoint of the simulation and the states at conjunction of both the satellite and the debris objects. In this case, our miss distance is very close to that predicted.

```
dMiss_self = vecnorm(mvr.XF_icrf - conj.X_icrf(:,:,1));
dMiss_debris = vecnorm(mvr.XF_icrf - conj.X_icrf(:,:,2));
disp(table(dMiss_self*1e-3, dMiss_debris*1e-3, sqrt(mvr.delD_R^2+mvr.delD_T^2), ...
  VariableNames=["Actual distance - self (km)","Miss distance - debris (km)","Predicted miss dis
```

Actual distance - self (km)	Miss distance - debris (km)	Predicted miss distance (km)
24.777	24.761	24.789

For visualization, we also retrieve the debris state from its TLE for the time vector output from the simulation.

```
mission.sc.StartTime = mission.startTime;
mission.sc.StopTime = mission.startTime+days(7);
pos_debris = zeros(3,length(ephPosICRF.Time));
vel_debris = zeros(3,length(ephPosICRF.Time));
for k = 1:length(ephPosICRF.Time)
  [pos_debris(:,k),vel_debris(:,k)] = states(mission.sat(2),ephPosICRF.Time(k)+mvr.mvrTime,Coord
end
```

Confirm the Collision Probability is Reduced

Check the new probability using `maxCollisionProbability2D`. Object 2's covariance is not changed. Object 1, the StarLink satellite, now has a different position so we should recompute the inertial covariance since the encounter frame has changed.

```
A = dcmicrf2lvlh(mvr.XF_icrf,mvr.VF_icrf);
mvr.cov1_icrf = A'*c0_lvlh*A; % convert LVLH covariance to inertial frame
```

The new probability is effectively zero, while the maximum probability is reduced to less than 1e-7 (depending on the hard-body radius selected above).

```
mvr.Pc = collisionProbability2D(mvr.XF_icrf,mvr.VF_icrf,mvr.cov1_icrf,...
  conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,conj.rSatellite);
mvr.Pcmax = maxCollisionProbability2D(mvr.XF_icrf,mvr.VF_icrf,mvr.cov1_icrf,...
  conj.X_icrf(:,:,2),conj.V_icrf(:,:,2),conj.cov2_icrf,conj.rSatellite);
disp(table( conj.Pc0, mvr.Pc, mvr.Pcmax, ...
  VariableNames=["Original Probability","New Probability",'New Max Probability']))
```

Original Probability	New Probability	New Max Probability
0.00057166	0	1.0442e-07

Using the rotation to the LVLH frame, we can check the miss distance in each direction. The cross-track or orbit normal direction offset is small but not nonzero because the orbits are not perfectly circular and the orbit perturbations.

```
dX_LVLH = A*(mvr.XF_icrf - conj.X_icrf(:,:,2));
disp(table( dX_LVLH(1)*1e-3, dX_LVLH(2)*1e-3, dX_LVLH(3)*1e-3, ...
  VariableNames=["Radial offset (km)","Along-Track Offset","Cross-Track Offset"]))
```

Radial offset (km)	Along-Track Offset	Cross-Track Offset
1.4578	-24.718	0.0069499

Visualize the Maneuver Results

View in a Scenario

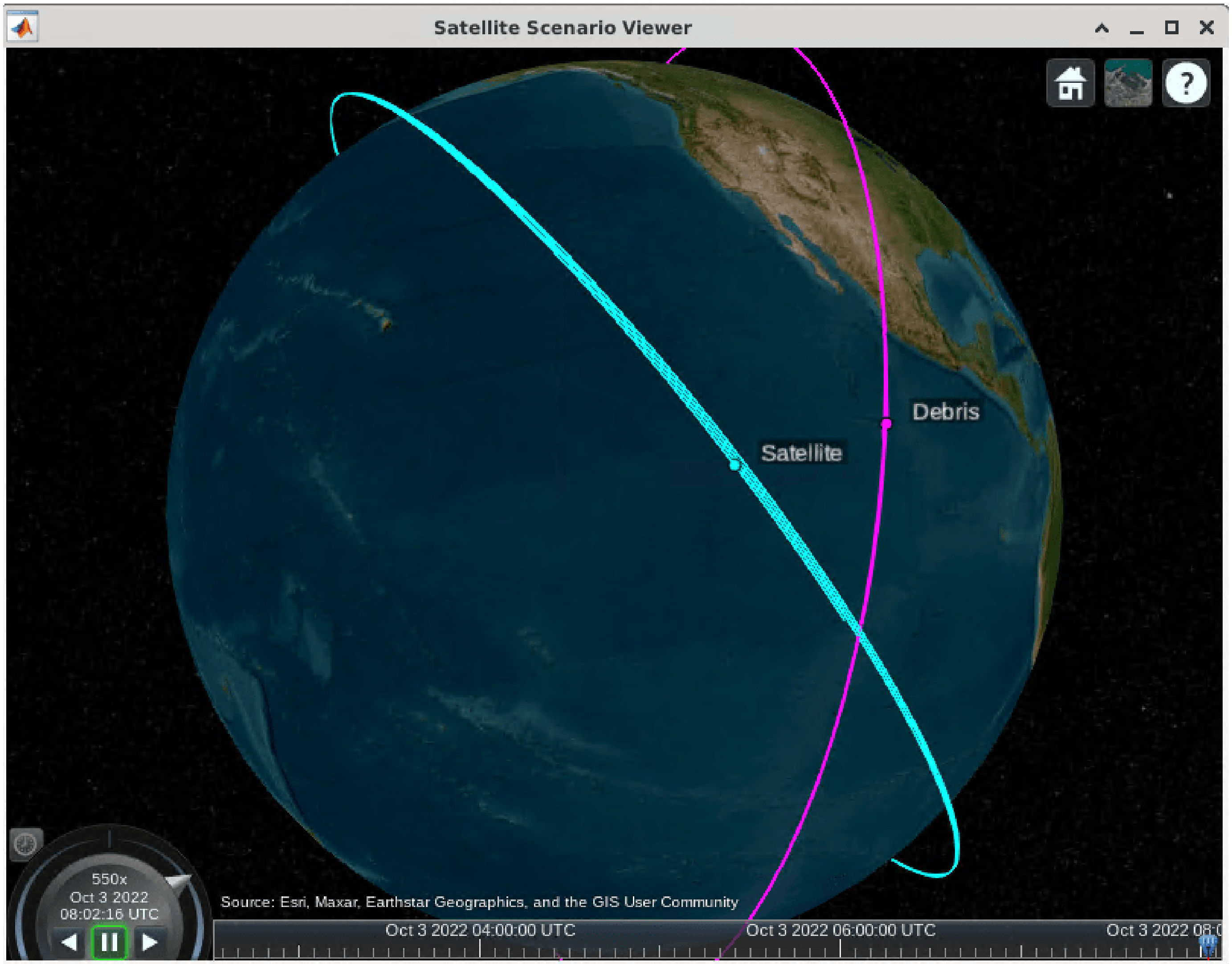
View the results in the scenario viewer. First, create a new scenario object associated with the maneuver:

```
mvr.sc = satelliteScenario;
```

Update the ephemeris start time with the datetime for loading it back into the satellite scenario. Create timetables for the debris object ephemeris obtained from the prior scenario's SGP4 propagator. Then, pass the scenario object to the viewer function to view the orbits.

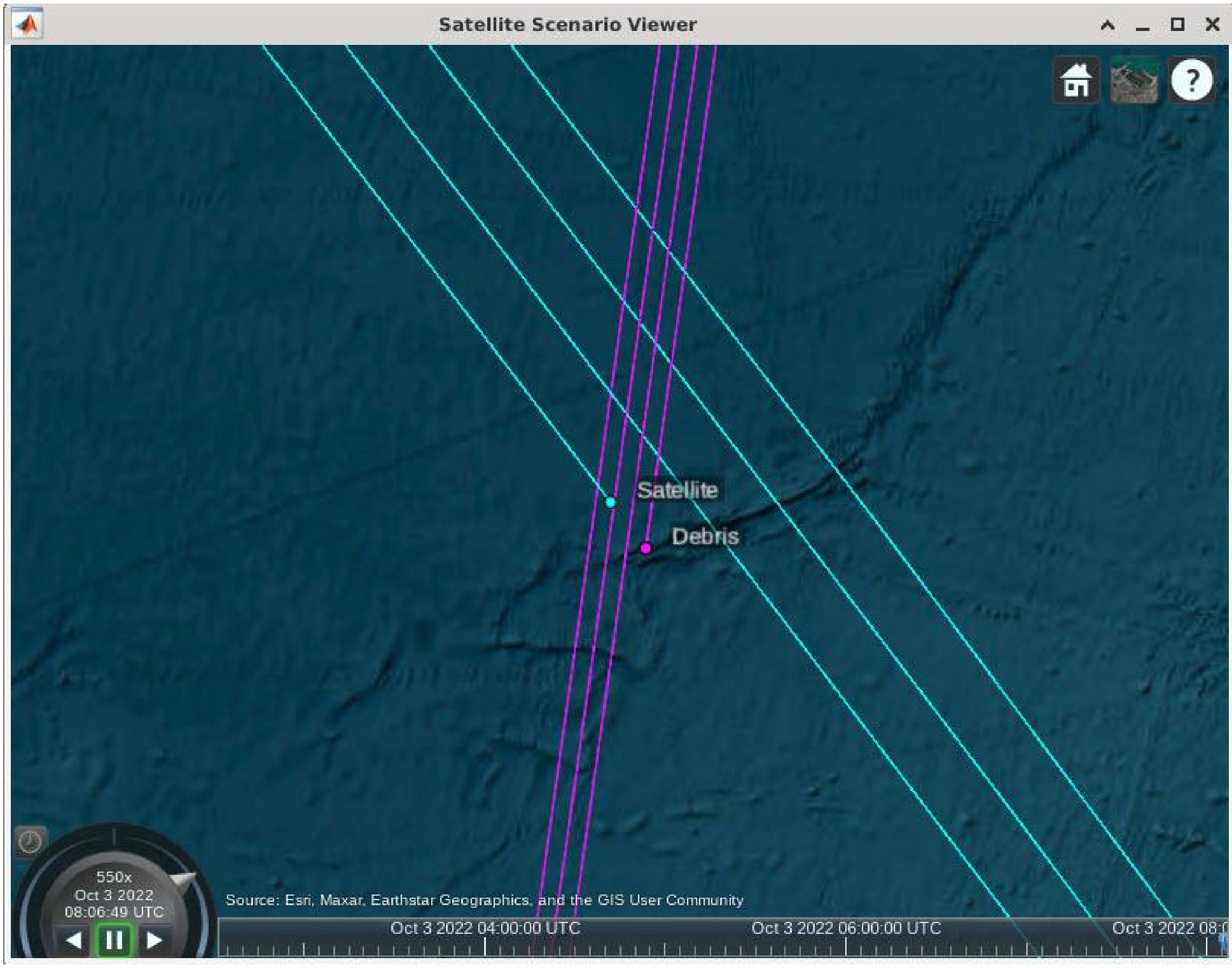
```
starlinkPos = ephPosICRF;
starlinkPos.Data = mvr.X_icrf';
starlinkPos.Properties.StartTime = mvr.mvrTime;
starlinkPos.Time.TimeZone = "UTC";
starlinkVel = ephVelICRF;
starlinkVel.Data = mvr.V_icrf';
starlinkVel.Properties.StartTime = mvr.mvrTime;
starlinkVel.Time.TimeZone = "UTC";

sat_n = satellite(mvr.sc, starlinkPos, starlinkVel, ...
    "CoordinateFrame", "inertial","Name","Satellite");

debrisPos = timetable(starlinkPos.Time,pos_debris'); % debris TLE position
debrisVel = timetable(starlinkPos.Time,vel_debris');

sat_d = satellite(mvr.sc, debrisPos, debrisVel, ...
    "CoordinateFrame", "inertial","Name","Debris");
sat_d.MarkerColor = [1 0.059 1];
sat_d.Orbit.LineColor = [1 0.059 1];

viewer = satelliteScenarioViewer(mvr.sc,"CurrentTime",mvr.sc.StopTime-days(5/1440));
camheight(viewer, 9.90e+06);
```

These screenshots, of the scenario, show that we have achieved an offset of our active satellites at the time when the debris is at its closest to the satellite orbit.




```
viewerZoom = satelliteScenarioViewer(mvr.sc,"CurrentTime",mvr.sc.StopTime);
camheight(viewerZoom, 1.0e+06);
```

This zoomed-in view at the end of the trajectories, which is the previous time of closest approach, shows that the satellite now lags the debris.



Relative State Plot

To view the relative state, we must rotate the difference in the states at every timestep. We do this in a loop for our two sets of timetables, the simulated data with and without the maneuver. We achieve a plot very similar to that of the reference [3].

```
xH = zeros(3,size(ephPosICRF,1));
for k = 1:size(ephPosICRF,1)
  r = no_mvr.X_icrf(:,k); % no maneuver
  v = no_mvr.V_icrf(:,k);
  A = dcmicrf2lvlh( r, v );  % Inertial to LVLH frame (RSW)
  xH(:,k) = A*(mvr.X_icrf(:,k) - r);
end
figure
plot(xH(1,:),xH(2,:))
grid on
grid minor
title('Relative State, Maneuver vs. No Maneuver')
```

```
xlabel('X in B-plane (radial) [m]')
ylabel('Y in B-plane (tangential) [m]')
```

We can plot the combined covariance by transforming the inertial covariances into the LVLH frame of the maneuvering satellite. To calculate the semi-major axes and rotation of the ellipse from the covariance, use the singular value decomposition with `svd()`. We plot the 1 and 2-sigma ellipses here.

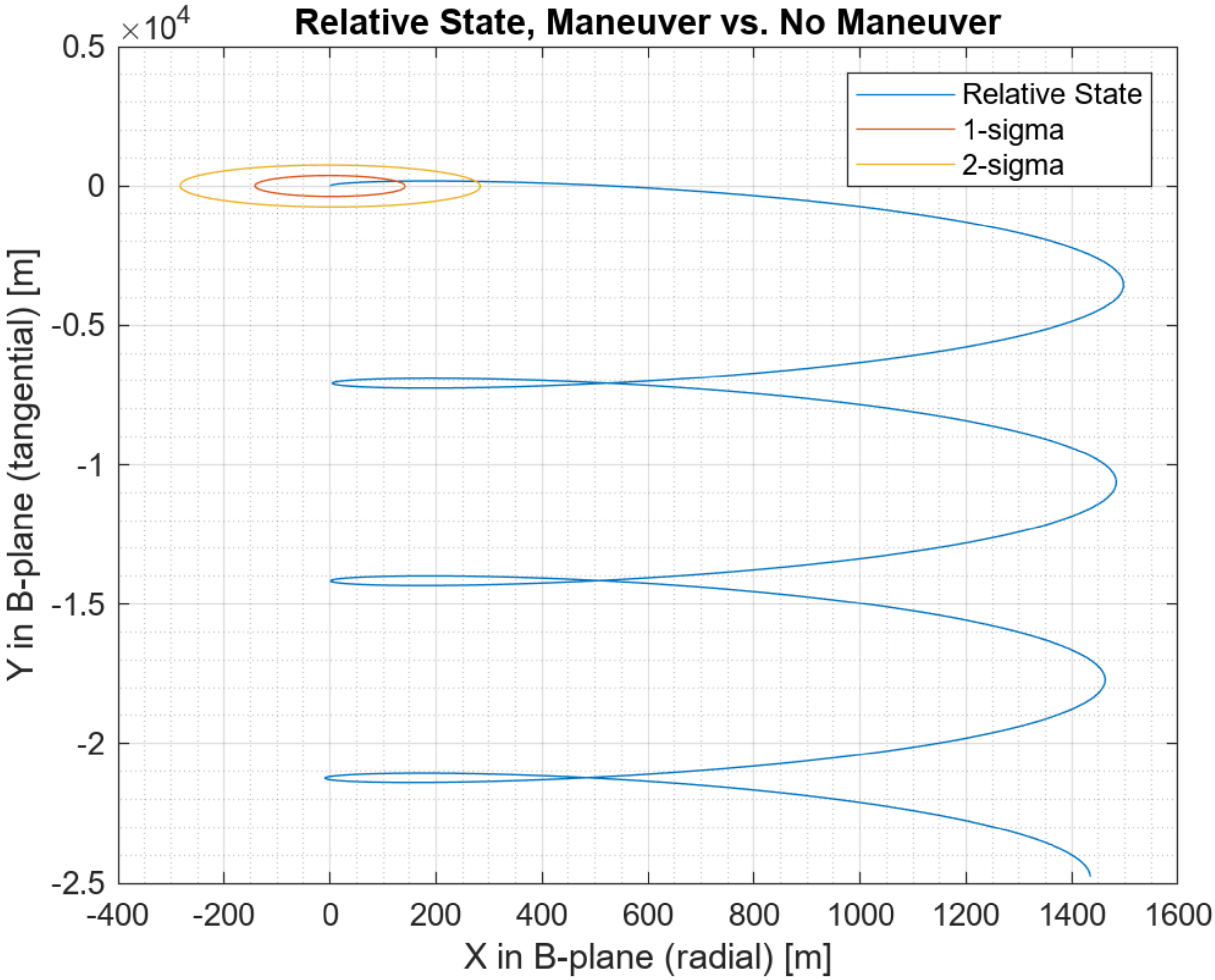
```
theta = linspace(0,2*pi);
A = dcmicrf2lvlh( mvr.X0_icrf, mvr.V0_icrf );
covSum = mvr.cov1_icrf+conj.cov2_icrf;
P_lvlh = A*covSum*A';
[u,s] = svd(P_lvlh(1:2,1:2));
sma_cov = sqrt(diag(s));
disp('Ellipse semi-major axes (m):')
```

```
Ellipse semi-major axes (m):
```

```
disp(sma_cov)
```

```
  373.7744
  141.4519
```

```
x = sma_cov(1)*cos(theta);
y = sma_cov(2)*sin(theta);
rE = u*[x;y];
hold on
plot(rE(1,:),rE(2,:))
plot(2*rE(1,:),2*rE(2,:))
legend('Relative State','1-sigma','2-sigma')
```



Note that in operational use, check the maneuver plan to make sure it does not cause any other potential conjunctions. In addition, the satellite may need to maneuver back to station. This is about the same magnitude maneuver as that executed, but needs to be planned closer to real-time to account for disturbances and maneuver errors.

References

1 NASA Spacecraft Conjunction Assessment and Collision Avoidance Best Practices Handbook, December 2020, NASA/SP-20205011318, **https://nodis3.gsfc.nasa.gov/OCE_docs/OCE_50.pdf**

2 Spaceflight Safety Handbook for Satellite Operators, Version 1.5, August 2020, 18th Space Control Squadron, www.space-track.org

3 Adia, S., "Conjunction Risk Assessment and Avoidance Maneuver Planning Tools." 6th International Conference on Astrodynamics Tools and Techniques, 2016. Available on ResearchGate.

4 Alfriend, K., Akella, M., Lee, D., Frisbee, J., Foster, J., Lee, D., and Wilkins, M. “Probability of Collision Error Analysis.” Space Debris Vol. 1 No. 1 (1999) pp. 21-35 Springer. DOI 10.1023/A:1010056509803

See Also

Blocks
Orbit Propagator

Objects
`satelliteScenario`

Custom Vertical Takeoff and Landing Example

This example highlights custom aircraft visualization capabilities by showing four example flights to destinations in the city of Boston. The vertical takeoff and landing (VTOL) locations are described by their latitude, longitude, and altitude (relative to WGS84). Each flight uses the custom aircraft mesh, each configured differently to illustrate possibilities available with the provided mesh.

The aircraft skeleton type is `Custom`. It has 57 bones, as shown in the following table.

Bone Index:	Bone Name:	Custom Mesh Part:
1	FixedWing	Entire aircraft
2	Engine1	Left inboard motor on the main wing
3	Engine1_Prop	Propeller on motor 1
4	Engine2	Left second from inboard motor on the main wing
5	Engine2_Prop	Propeller on motor 2
6	Engine3	Left third from inboard motor on the main wing
7	Engine3_Prop	Propeller on motor 3
8	Engine4	Left fourth from inboard motor on the main wing
9	Engine4_Prop	Propeller on motor 4
10	Engine5	Left fifth from inboard motor on the main wing
11	Engine5_Prop	Propeller on motor 5
12	Engine6	Left outboard motor on the main wing
13	Engine6_Prop	Propeller on motor 6
14	Engine7	Left inboard motor on the front wing
15	Engine7_Prop	Propeller on motor 7
16	Engine8	Left outboard motor on the front wing
17	Engine8_Prop	Propeller on motor 8
18	Engine9	Right inboard motor on the main wing
19	Engine9_Prop	Propeller on motor 9
20	Engine10	Right second from inboard motor on the main wing
21	Engine10_Prop	Propeller on motor 10
22	Engine11	Right third from inboard motor on the main wing
23	Engine11_Prop	Propeller on motor 11
24	Engine12	Right fourth from inboard motor on the main wing
25	Engine12_Prop	Propeller on motor 12
26	Engine13	Right fifth from inboard motor on the main wing
27	Engine13_Prop	Propeller on motor 13
28	Engine14	Right outboard motor on the main wing
29	Engine14_Prop	Propeller on motor 14
30	Engine15	Right inboard motor on the front wing
31	Engine15_Prop	Propeller on motor 15
32	Engine16	Right outboard motor on the front wing
33	Engine16_Prop	Propeller on motor 16
34	Wing1	Main wing
35	Wing1_Aileron_L	Left aileron on the main wing
36	Wing1_Aileron_R	Right aileron on the main wing
37	Wing1_Flap_L	Left flap on the main wing
38	Wing1_Flap_R	Right flap on the main wing
39	Wing1_Spoiler_L	Left spoiler on the main wing
40	Wing1_Spoiler_R	Right spoiler on the main wing
41	Wing2	Front wing
42	Wing2_Flap_L	Left flap on the front wing
43	Wing2_Flap_R	Right flap on the front wing
44	Rudder_L	Bottom half of the rudder
45	Rudder_R	Top half of the rudder
46	HorizStab	Horizontal stabilizer
47	HorizStab_Elevator_L	Left half of the elevator
48	HorizStab_Elevator_R	Right half of the elevator
49	NoseGear	Nose gear
50	NoseGear_Wheel	Nose gear wheel
51	NoseGearDoor	Nose gear door
52	MainGear_L	Left main gear
53	MainGear_L_Wheel	Left main gear wheel
54	MainGear_R	Right main gear
55	MainGear_R_Wheel	Right main gear wheel
56	MainGearDoor_L	Left main gear door

The translation and rotation structure `signals.values` field indexes these bones from 1 to 57. Custom aircraft mesh bones are oriented with their primary axis vertical, except for rotating meshes. Engine and propeller bones face forward, and wheel bones face to the right. The bone *y*-axis is primary.

Set Up the Example

To create the variables used by the Simulink model `CustomVTOLExample.slx:`

1. Start MATLAB with administrative privileges.
2. Run this live script. Translation and rotation structures are created for the From Workspace blocks in the model.
3. In the model, customize the Simulation 3D Scene Configuration block with your Cesium Ion® token. For more information, see "Visualize with Cesium".

Boston Tour Mission

The mission is to make four flights to destinations in the city of Boston. The aircraft latitude, longitude, and altitude describe the takeoff and landing locations.

```
npts = 5;  % number of flights plus one
takeoffpts = zeros(npts, 3);
takeoffpts(1,:) = [42.329545, -71.049227, -24.80];  % Thomas Viens Little League fields
takeoffpts(2,:) = [42.343470, -71.049331, -23.50];  % Boston Convention & Exhibit Center
takeoffpts(3,:) = [42.333805, -71.071351, -24.20];  % Boston City Hospital helipad
takeoffpts(4,:) = [42.354255, -71.067202, -23.35];  % Boston Common baseball diamond
takeoffpts(5,:) = [42.371606, -71.057163, -25.90];  % USS Constitution Wharf
```

Latitude and longitude values can be found from a mapping application. The altitude is the height above the WGS84 ellipsoid model of the Earth at the given latitude and longitude. Experimentally find the altitude by first using a value that places the aircraft above ground. Run the model. Note the altitude sensor reading at that takeoff or landing point. Then correct the altitude value above by subtracting the altitude sensor reading from it to place the aircraft on the terrain.

Determining the flight time between these locations requires the distance. The distance between latitude and longitude pairs can be found in various ways. This example uses the `geoTrajectory` object in the helper function `latlondist`. The true course (i.e., azimuth) angle between takeoff and landing locations is obtained at the same time.

```
ac.Distance = zeros(1, npts-1);
ac.Course = zeros(1, npts-1);
for i = 1:npts-1
    [ac.Distance(i), ac.Course(i)] = latlondist(takeoffpts(i,1), ...
        takeoffpts(i,2), takeoffpts(i+1,1), takeoffpts(i+1,2));
end
```

Conditions

Set up the time conditions of the flight.

The performance of the aircraft impacts the flight time. Set the desired flight speed, rate of climb, acceleration, and cruising altitude. The rate of climb is used as the rate of descent as well. The International System of Units (SI) are used throughout this example.

```
ac.FlightAlt = 100;
ac.RateOfClimb = 10;
ac.FlightSpeed = 103;
ac.Accel = 10;
```

To reorient and reconfigure in between flights requires a setup time that must be five seconds or greater.

These events happen during this time:

1. Transition from takeoff to cruise mode.
2. Reconfiguration and reorientation to the new course.
3. Transition back to takeoff mode.

To allow adequate time for reconfiguring and reorientation, the time spent by the two mode transitions, `2*act.Trans`, must be less than `act.Config`.

```
act.Config = 6;
act.Trans = 2;
if act.Config < 5
    act.Config = 5;
end
if act.Config-2*act.Trans < 1
    act.Config = 2*act.Trans + 1;
end
```

Calculate the total mission time. The local function, `flighttimes,` returns both the total and the segment times.

```
[act.Total, act.Segments] = flighttimes(ac.Distance, act.Config, ...
    ac.FlightAlt, ac.RateOfClimb, ac.FlightSpeed, ac.Accel);
```

Create convenient segment indices for each of the four flights.

```
j = 0:npts-2;
isegment.Config = [1 + 6*j, length(act.Segments)];
isegment.Takeoff = 2 + 6*j;
isegment.Accel =   3 + 6*j;
isegment.Land =    6 + 6*j;
```

Create a running total time.

```
act.Running = zeros(1, length(act.Segments));
for i = 2:length(act.Segments)
    act.Running(i) = act.Running(i-1) + act.Segments(i-1);
end
```

Initialization

Initialize the translation and rotation structures with time for the From Workspace blocks. Use a fixed time step of one-tenth of a second.

```
nseconds = floor(act.Total);
fprintf('The model run time is %d seconds.\n', nseconds)
```

```
The model run time is 229 seconds.
```

```
samplerate = 0.1;
nsteps = 1 + floor(nseconds/samplerate);
translation.time = samplerate*double(0:(nsteps-1));
translation.time = translation.time';
rotation.time = translation.time;
```

Custom aircraft dimensions are as follows:

```
nrows = 57;
ncols = 3;
translation.signals.dimensions = [nrows ncols];
rotation.signals.dimensions = [nrows ncols];
```

Initialize translations and rotations to zero for all of the bones and time steps.

```
translation.signals.values = zeros(nrows, ncols, nsteps);
rotation.signals.values = zeros(nrows, ncols, nsteps);
```

As the flight progresses, the example adjusts the altitude. The model sets the origin to the latitude, longitude, and height of the first takeoff point.

Clean up temporary variables.

```
clear i j samplerate nrows ncols npts
```

First Flight Configuration: Wings Rotate for Takeoff and Landing

For the first flight, the example uses the default aircraft configuration. During the last two seconds of the initial configuration time, to prepare for vertical takeoff, the example rotates the two wings to vertical.

```
icfg = 1;     % configuration and flight index, [1:4]
is1 = isegment.Takeoff(icfg);   % trunning(is1) is time at start of takeoff
tfinal = act.Running(isegment.Config(icfg+1)) + act.Segments(is1);  % time at end of config after
[translation, rotation] = acsetup(1, icfg, act.Running(is1)-act.Trans, ...
    act.Trans, tfinal, 90-ac.Course(icfg), translation, rotation);  % use UE heading
is2 = isegment.Accel(icfg);  % trunning(is2) is time at end of takeoff
[translation, rotation] = acconfig(icfg, act.Running(is1)-act.Trans, ...
    act.Trans, act.Running(is2), "takeoff", translation, rotation);  % hold through ascent
```

To take off, spin the propellers and ascend to cruise altitude.

```
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), ac.FlightAlt, translation, rotation);
```

The cruise portion of the flight begins with an acceleration to cruise speed, the first portion of which includes a transition from vertical to horizontal thrust configuration. Similarly, the last portion of the cruise includes a transition back to vertical thrust.

```
is1 = is2;  % trunning(is1) is time at start of acceleration to cruise
is2 = isegment.Land(icfg);  % trunning(is2) is time at end of deceleration from cruise
tmodetrans = min(6, act.Segments(is1));  % transition between vertical and horizontal flight mode
[translation, rotation] = accruise(icfg, act.Running(is1:is2+1), ...
    tmodetrans, ac.FlightSpeed, ac.Course(icfg), translation, rotation);
```

Note: The transition to and from vertical and horizontal (wing-supported) flight is not modeled realistically. The purpose of this example is to illustrate the visualization capabilities of the `Custom` aircraft type, not to show VTOL aircraft dynamics.

The descent is the reverse of the ascent.

```
is1 = is2;  % trunning(is1) is time at start of landing
is2 = isegment.Config(icfg+1);  % trunning(is2) is time at end of landing
hdescent = takeoffpts(icfg+1, 3) - ac.FlightAlt - takeoffpts(icfg, 3);  % negative
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), hdescent, translation, rotation);
```

Finish the first flight by returning the aircraft to its default mode, where the propellers all face forward.

```
is1 = is2;  % trunning(is1) is time at start of config after landing
[translation, rotation] = acconfig(icfg, act.Running(is1), ...
    act.Trans, tfinal, "cruise", translation, rotation);
```

Second Flight Configuration: Motor Pods Rotate for Takeoff and Landing

For the second flight configuration, only the motor pods rotate to vertical for takeoff and landing. Both the main and the front wing stay fixed. However, their flaps are deployed to assist in vertical flight.

Otherwise, the steps are the same as for the first flight. The reconfiguration happens in the middle of the grounded time.

```
icfg = 2;  % configuration and flight #2
is1 = isegment.Config(icfg);  % trunning(is1) is time at start of this tconfig period
tfinal = act.Running(isegment.Config(icfg+1)) + act.Segments(is1);  % time at end of config after
[translation, rotation] = acsetup(icfg-1, icfg, act.Running(is1)+act.Trans, ...
    act.Config-2*act.Trans, tfinal, 90-ac.Course(icfg), translation, rotation);  % UE heading
is1 = isegment.Takeoff(icfg);
is2 = isegment.Accel(icfg);  % trunning(is2) is time at end of takeoff
[translation, rotation] = acconfig(icfg, act.Running(is1)-act.Trans, ...
    act.Trans, act.Running(is2), "takeoff", translation, rotation);  % hold through ascent
```

Takeoff.

```
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), ac.FlightAlt, translation, rotation);
```

Cruise.

```
is1 = is2;  % trunning(is1) is time at start of acceleration to cruise
is2 = isegment.Land(icfg);  % trunning(is2) is time at end of deceleration from cruise
tmodetrans = min(6, act.Segments(is1));  % transition between vertical and horizontal flight mode
[translation, rotation] = accruise(icfg, act.Running(is1:is2+1), ...
    tmodetrans, ac.FlightSpeed, ac.Course(icfg), translation, rotation);
```

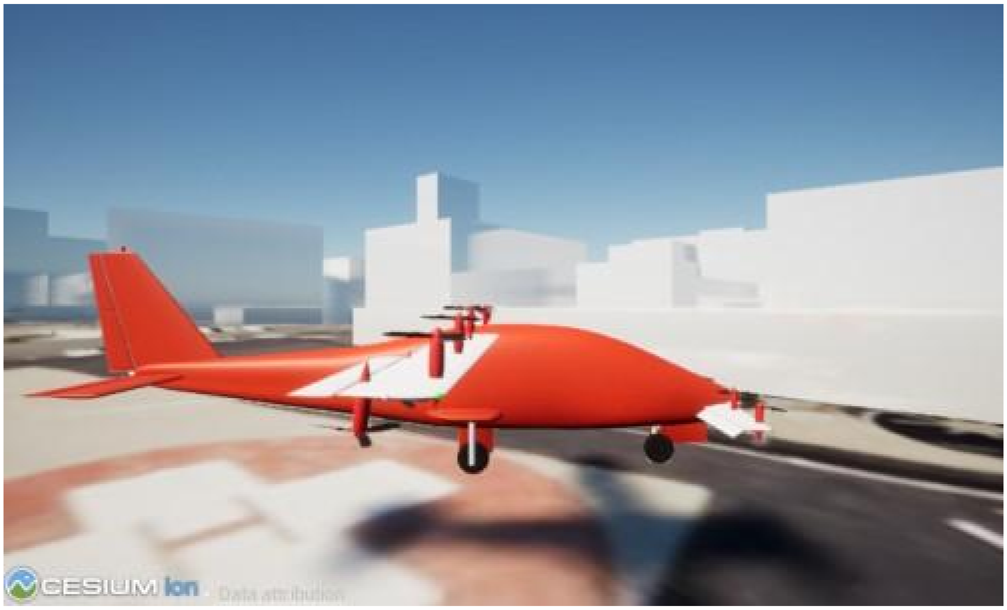
Descend to land.

```
is1 = is2;  % trunning(is1) is time at start of landing
is2 = isegment.Config(icfg+1);  % trunning(is2) is time at end of landing
hdescent = takeoffpts(icfg+1, 3) - ac.FlightAlt - takeoffpts(icfg, 3);  % negative
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), hdescent, translation, rotation);
```

Reconfigure.

```
is1 = is2;  % trunning(is1) is time at start of config after landing
[translation, rotation] = acconfig(icfg, act.Running(is1), act.Trans, tfinal, "cruise", ...
    translation, rotation);
```

Third Flight Configuration: Motor Pods Rotate for Takeoff and Landing with Six Pusher Props



For the third flight configuration, move half of the main wing motors behind the wing to act as pushers. As with the second configuration, only the motor pods rotate to vertical for takeoff and landing.

```
icfg = 3;  % configuration and flight #3
is1 = isegment.Config(icfg);  % trunning(is1) is time at start of this tconfig period
tfinal = act.Running(isegment.Config(icfg+1)) + act.Segments(is1);  % time at end of config after
[translation, rotation] = acsetup(icfg-1, icfg, act.Running(is1)+act.Trans, ...
    act.Config-2*act.Trans, tfinal, 90-ac.Course(icfg), translation, rotation);  % UE heading
is1 = isegment.Takeoff(icfg);
is2 = isegment.Accel(icfg);  % trunning(is2) is time at end of takeoff
```

```
[translation, rotation] = acconfig(icfg, act.Running(is1)-act.Trans, ...
    act.Trans, act.Running(is2), "takeoff", translation, rotation);  % hold through ascent
```

Takeoff.

```
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), ac.FlightAlt, translation, rotation);
```

Cruise.

```
is1 = is2;  % trunning(is1) is time at start of acceleration to cruise
is2 = isegment.Land(icfg);  % trunning(is2) is time at end of deceleration from cruise
tmodetrans = min(6, act.Segments(is1));  % transition between vertical and horizontal flight mode
[translation, rotation] = accruise(icfg, act.Running(is1:is2+1), ...
    tmodetrans, ac.FlightSpeed, ac.Course(icfg), translation, rotation);
```

Descend to land.

```
is1 = is2;  % trunning(is1) is time at start of landing
is2 = isegment.Config(icfg+1);  % trunning(is2) is time at end of landing
hdescent = takeoffpts(icfg+1, 3) - ac.FlightAlt - takeoffpts(icfg, 3);  % negative
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), hdescent, translation, rotation);
```

Reconfigure.

```
is1 = is2;  % trunning(is1) is time at start of config after landing
[translation, rotation] = acconfig(icfg, act.Running(is1), act.Trans, tfinal, "cruise", ...
    translation, rotation);
```

Fourth Flight Configuration: Front Wing and Main Wing Motor Pods Rotate for Takeoff and Landing with Two Fixed Cruise Motors

For the fourth flight configuration, move two of the wing motors to the horizontal stabilizer. They remain fixed in the forward-facing position. In this configuration, the main wing does not rotate to vertical, but the front wing does.

```
icfg = 4;  % configuration and flight #4
is1 = isegment.Config(icfg);  % trunning(is1) is time at start of this tconfig period
tfinal = translation.time(end);  % tfinal for last flight
[translation, rotation] = acsetup(icfg-1, icfg, act.Running(is1)+act.Trans, ...
    act.Config-2*act.Trans, tfinal, 90-ac.Course(icfg), translation, rotation);  % UE heading
is1 = isegment.Takeoff(icfg);
is2 = isegment.Accel(icfg);  % trunning(is2) is time at end of takeoff
[translation, rotation] = acconfig(icfg, act.Running(is1)-act.Trans, ...
    act.Trans, act.Running(is2), "takeoff", translation, rotation);  % hold through ascent
```

Takeoff.

```
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), ac.FlightAlt, translation, rotation);
```

Cruise.

```
is1 = is2;  % trunning(is1) is time at start of acceleration to cruise
is2 = isegment.Land(icfg);  % trunning(is2) is time at end of deceleration from cruise
tmodetrans = min(6, act.Segments(is1));  % transition between vertical and horizontal flight mode
[translation, rotation] = accruise(icfg, act.Running(is1:is2+1), ...
    tmodetrans, ac.FlightSpeed, ac.Course(icfg), translation, rotation);
```

Descend to land.

```
is1 = is2;  % trunning(is1) is time at start of landing
is2 = isegment.Config(icfg+1);  % trunning(is2) is time at end of landing
hdescent = takeoffpts(icfg+1, 3) - ac.FlightAlt - takeoffpts(icfg, 3);  % negative
[translation, rotation] = actakeoff(icfg, act.Running(is1), ...
    act.Running(is2), hdescent, translation, rotation);
```

Reconfigure.

```
is1 = is2;  % trunning(is1) is time at start of config after landing
[translation, rotation] = acconfig(icfg, act.Running(is1), act.Trans, tfinal, "cruise", ...
    translation, rotation);
```

Clean up temporary variables.

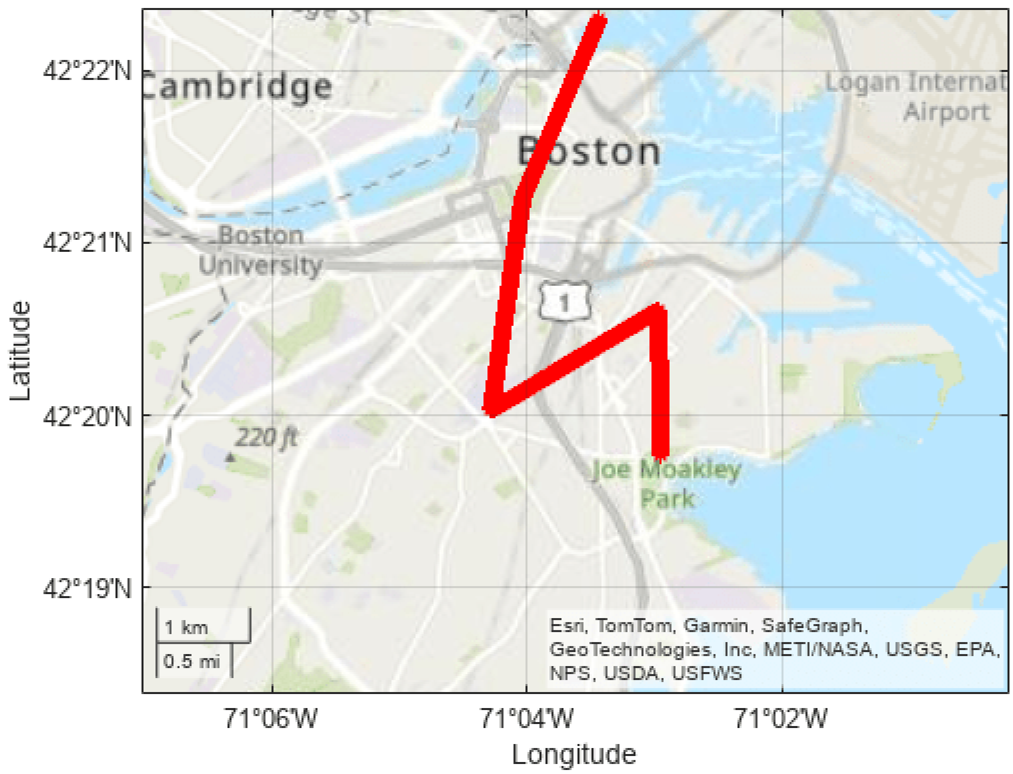
```
clear isegment icfg is1 is2 hdescent tmodetrans tfinal
```

Plot Flight Tracks

To plot the flight ground tracks, use the `flat2lla` function to convert from the north-east-down (NED) coordinate frame to geodetic coordinates. Plot the latitude and longitude points on a topographic map. Note that the translation and rotation arrays for Unreal Engine® have the *x*-axis pointing East and the *y*-axis pointing South.

```
lat = zeros(nsteps, 1);
lon = zeros(nsteps, 1);
lat(1) = takeoffpts(1,1);
lon(1) = takeoffpts(1,2);
for i = 2:nsteps
    xNorth = -translation.signals.values(1, 2, i);
    yEast = translation.signals.values(1, 1, i);
    lla = flat2lla([xNorth, yEast, 0], [takeoffpts(1,1), takeoffpts(1,2)], 0, 0);
    lat(i) = lla(1);
    lon(i) = lla(2);
end
```

```
geobasemap topographic
geoplot(lat, lon, '*r')
[latlims, lonlims] = geolimits;
dlat = 0.5*(latlims(2) - latlims(1));
geolimits([latlims(1)-dlat, latlims(2)], [lonlims(1)-0.5*dlat, lonlims(2)+0.5*dlat])
```



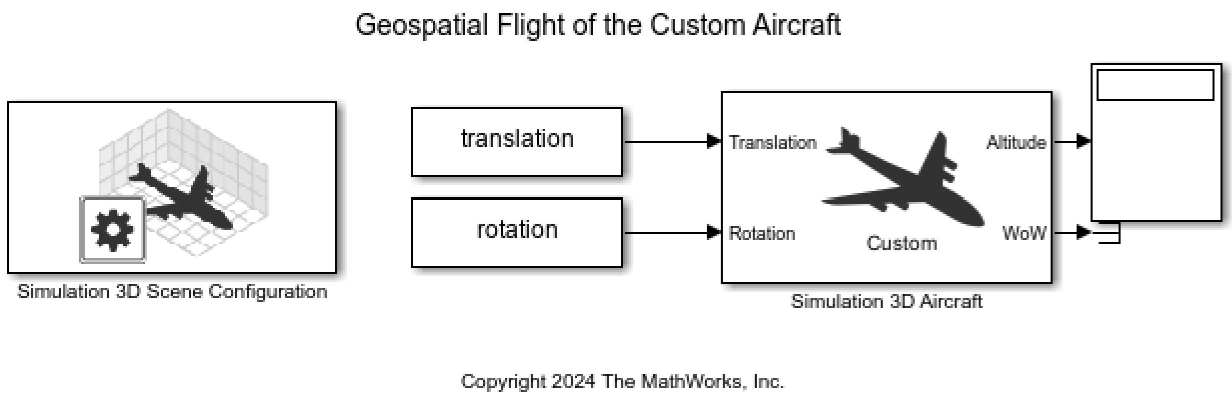
Clean up temporary variables.

```
clear i xNorth yEast lat lon nsteps latlims lonlims dlat
```

Run

Now that the translation and rotation structures have been fully calculated, open the Simulink® model and set the stop time.

```
open_system("CustomVTOLExample.slx");
set_param("CustomVTOLExample", "StopTime", num2str(nseconds));
```



Click the **Run** button to view the flights.

If takeoff and landing locations or performance numbers are changed, rerun this live script.

Helper Functions

Besides the two helper functions below, this example uses eight other functions: `acconfig`, `accruise`, `acldggear`, `acsetup`, `actakeoff`, `acwingflaps`, `geotransform`, and `timeindices`.

```
function [distance, azimuth] = latlondist(lat1, lon1, lat2, lon2)
%LATLONDIST Calculate the distance and azimuth angle between a pair
%of locations, defined by their latitude and longitude.
%
% Use Sensor Fusion's "geoTrajectory" system object to make the
% calculations.
%
% Input Variables:
%   lat1, lon1 = starting location
%   lat2, lon2 = ending location
%
% Output Variables:
%   distance = distance in meters between starting and ending locations
%   azimuth = azimuth in degrees of ending location with respect to the
%             starting location

traveltime = [0 1];  % 1 second trajectory so velocity equals distance
samplerate = 0.5;      % 3 samples (doesn't affect accuracy)
trajectory = geoTrajectory([lat1 lon1, 0; lat2, lon2, 0], traveltime, ...
    'SampleRate', samplerate);

% Use lookupPose to get velocity (m/s) at end of trajectory
[~, ~, velocity, ~, ~, ~] = lookupPose(trajectory, 1);
distance = sqrt(velocity(1)^2 + velocity(2)^2);

% Get azimuth from the Course property
azimuth = trajectory.Course(end);
while azimuth < 0  % add 360 degrees if negative
    azimuth = 360 + azimuth;
end
end
```

```
function [ttotal, tsegments] = flighttimes(distances, tconfig, ...
    altitude, acroc, acspeed, acaccel)
%FLIGHTTIMES Calculate flight segment times.
% Find the time for each segment of a VTOL mission. VTOL flights have
% 1 + 6N segments for "N" total flights: configure, climb, accelerate,
% cruise, decelerate, descend, reconfigure, etc. Note that the wind is
% assumed to be calm.
%
% Input Variables:
%   distances = 1-by-N vector of flight distances (m)
%   tconfig = time before, between, and after each flight (s)
%   altitude = cruise altitude above the takeoff point (m)
%   acroc = aircraft rate of climb (m/s)
%   acspeed = aircraft cruise flight speed (m/s)
%   acaccel = aircraft rate of acceleration in level flight (m/s^2)
%
% Output Variables:
%   ttotal = total mission time (s)
%   tsegments = mission time for each segment (s)

N = length(distances);
tsegments = zeros(1, N);

is = 1;  % segment index
tsegments(is) = tconfig;
is = is + 1;
for i = 1:N
    tsegments(is) = altitude / acroc;
    is = is + 1;
    tsegments(is) = acspeed / acaccel;
    is = is + 1;
    % Get distance flown during acceleration and deceleration segments
    acdcdist = distances(i) - acspeed*tsegments(is-1);
    if acdcdist < 0
        error("Error: Zero time remaining for cruise segment!");
    end
    tsegments(is) = acdcdist / acspeed;
    is = is + 1;
    tsegments(is) = tsegments(is-2);  % deceleration = acceleration
    is = is + 1;
    tsegments(is) = tsegments(is-4);  % descent = climb
    is = is + 1;
    tsegments(is) = tconfig;
    is = is + 1;
end
ttotal = sum(tsegments);
end
```

See Also

Blocks
Simulation 3D Scene Configuration | Simulation 3D Aircraft

Tools
Custom

Related Examples

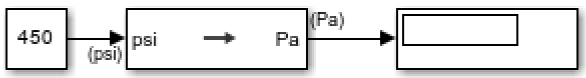
- "Visualize with Cesium" on page 2-42

Convert Pressure

Convert pressure from psi to Pa.

Open the model.

```
mdl = "ConvertPressurePsiToPa";
open_system(mdl);
```



Simulate the model.

```
sim(mdl);
```

Computation of Thrust and Torque Coefficients Using Rotor Block

This example shows how the Rotor block in Aerospace Blockset™ can be used to compute the thrust and torque coefficients for an isolated rotor. This example compares the results of this computation with the values obtained using Blade Element Momentum Theory (BEMT) and where available, the experimental results from literature [1]. This example considers the variation of lift and drag coefficient with angle of attack, and the effect of profile drag in the BEMT computation. The results from the Rotor block are also compared with experimental results from literature.

In the BEMT computation, the `fsolve` (Optimization Toolbox) function in Optimization Toolbox™ is used to solve for the average inflow through the rotor disc.

Rotor model

The rotor model considered in this example is taken from [2]. These parameters are saved in `Ref1Nb2Data.mat` as the structure array `rotorParams`:

- `Nb:` Number of blades in the rotor
- `R:` Radius of the blade, in meters
- `e:` Blade hinge-offset, in meters, measured from center of rotation
- `rc:` Root cutout, in meters
- `c:` Blade chord, in meters
- `rpm:` Rotor revolutions per minute
- `a:` blade section lift curve slope, in per rad

The root cutout is the inboard region of the blade where the rotor hub, hinges and the bearings are attached. This region has high drag coefficient and does not contribute to the lift.

Load the `Ref1Nb2Data.mat` file to add these parameters to the workspace:

```
refData = load('Ref1Nb2Data.mat');
Nb = refData.rotorParams.Nb;
radius = refData.rotorParams.R;
hingeOffset = refData.rotorParams.e;
rootCutout = refData.rotorParams.rc;
chord = refData.rotorParams.c;
OmegaRPM = refData.rotorParams.rpm;
Omega = convangvel(OmegaRPM, 'rpm' ,'rad/s');
sigma = Nb*chord/pi/radius; % rotor solidity
clalpha = refData.rotorParams.a;
```

Convert the reference rotor rotational velocity in RPM (Revolutions Per Minute) to rad/s using the `convangvel` function in Aerospace Toolbox. The rotor solidity σ is the ratio of the blade area (N_bcR for constant chord blades) to the rotor disc area (πR^2).

$$\sigma = \frac{N_bcR}{\pi R^2} = \frac{N_bc}{\pi R}$$

The lift and drag coefficient data [2] corresponding to the NACA0015 blade airfoil is saved in `NACA0015Data.mat.`

This file contains this date:

- `theta_NACA0015`: Array of angle of attack values in degrees at which lift/drag coefficient data is available
- `CL_NACA0015`: Array of lift coefficient values
- `CD_NACA0015`: Array of drag coefficient values

Load the data in the `NACA0015Data.mat` file.

```
airfoilData = load('NACA0015data.mat');
```

Blade Element Momentum Theory (BEMT)

Blade Element Momentum Theory combines the Simple Momentum Theory (SMT) and the Blade Element Theory (BET) [1].

Simple Momentum Theory

In SMT, the rotor is an actuator disc that can support a pressure difference and accelerate the air through the disc. It is based on basic conservation laws of fluid mechanics.

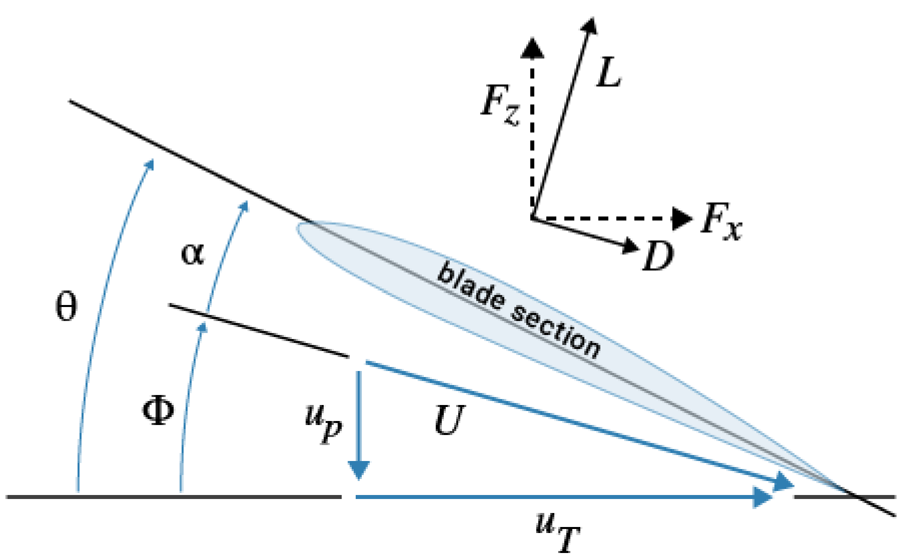
Using this theory, the rotor thrust (T) is related to the induced velocity (ν) at the rotor disc by,

$$T = 2\rho A\nu^2$$

where ρ is the density of air and $A$is the rotor disc area (πR^2).

Blade Element Theory

In BET, the blade is divided into airfoil sections capable of generating aerodynamic forces. This figure shows the blade section aerodynamics.



where,

- θ: pitch angle of blade

- ϕ: inflow angle
- α: aerodynamic angle of attack
- L: lift force at the blade section
- D: drag force at the blade section
- F_z: net force perpendicular to the disc plane
- F_x: net force tangential to the disc plane
- U_P: perpendicular air velocity seen by the blade
- U_T: tangential air velocity seen by the blade
- U: resultant velocity

and

$$U = \sqrt{U_T^2 + U_P^2}$$

$$\phi = \tan^{-1}\frac{U_P}{U_T}$$

$\alpha = \theta - \phi$.

The sectional lift and drag forces are defined as

$$L = \frac{1}{2}\rho U^2 c c_l$$

$D = \frac{1}{2}\rho U^2 c c_d$,

where ρ is the air density. c_l and c_d are the sectional lift and drag coefficients, and are typically functions of the angle of attack, Mach number and other parameters. In this example, we consider the coefficients to be functions of angle of attack alone (based on the available airfoil data).

The aerodynamic forces perpendicular and parallel to the disc plane are defined as

$$F_z = L\cos\phi - D\sin\phi$$

$$F_x = L\sin\phi + D\cos\phi$$

In hover, $U_P = \nu$, the induced velocity, and $U_T = \Omega y$, where Ω is the rotational velocity of the rotor and y is the dimensional radial location. The non-dimensional radial location is represented by $r = \frac{y}{R}$.

Small angle approximations can be applied [1], to obtain

$$U \approx U_T = \Omega y, \phi \approx \frac{U_P}{U_T} = \frac{\nu}{\Omega y} = \frac{\lambda}{r}$$

$F_z = L, F_x = L\phi + D$.

Here, $\lambda = \frac{\nu}{\Omega R}$ is the non-dimensional induced velocity or inflow ratio.

The elemental thrust and torque can now be defined as

$$\Delta T = N_b F_z \Delta r \approx N_b L \Delta r$$

$$\Delta Q = N_b F_x r R \Delta r \approx N_b (L\phi + D) r R \Delta r$$

Non-dimensionalizing the quantities to obtain the thrust and torque coefficients,

$$\Delta C_T = \frac{\Delta T}{\rho \pi R^2 (\Omega R)^2} = \frac{N_b L \Delta r}{\rho \pi R^2 (\Omega R)^2} = \frac{\sigma c_l r^2}{2} \Delta r$$

$$\Delta C_Q = \frac{\Delta Q}{\rho \pi R^3 (\Omega R)^2} = \frac{N_b (L\phi + D) r R \Delta r}{\rho \pi R^3 (\Omega R)^2} = \lambda \Delta C_T + \frac{\sigma c_d r^3}{2} \Delta r$$

Combining Momentum and Blade Element theories

Considering a rotor in hover, based on Momentum theory, the incremental thrust ΔT for a rotor annulus of width $\Delta y = R \Delta r$ at radial position r is

$$\Delta T = 2 \rho d A \nu^2 = 4 \rho \pi r R^2 \Delta r.$$

The corresponding non-dimensional thrust coefficient is

$$\Delta C_T = \frac{\Delta T}{\rho \pi R^2 (\Omega R)^2} = 4 \lambda^2 r \Delta r.$$

To account for the blade tip-loss effects, the Prandtl tip loss function[2] F can be is included as

$$\Delta C_T = 4 F \lambda^2 r \Delta r$$

where $F = \frac{2}{\pi} exp(-f), f = \frac{N_b}{2} \frac{1 - r}{r\phi}$.

Based on BET, the incremental thrust coefficient for the annulus is

$$\Delta C_T = \frac{\sigma c_l}{2} r^2 \Delta r.$$

The incremental torque coefficient ΔC_Q is

$$\Delta C_Q = \lambda \Delta C_T + \frac{\sigma c_d}{2} r^3 \Delta r.$$

Since the lift and drag coefficients depend on the local angle of attack, α, using interpolation on the available reference data, the coefficient values can be obtained at the required α.

The angle of attack is computed as

$$\alpha = \theta - \phi = \theta - \frac{\lambda}{r}.$$

Here, θ is the blade pitch angle at a radial location and depends on the blade twist and the pitch input. The reference blade considered in this example is untwisted and as pitch input, only the collective pitch(θ_0) variation is considered.

Hence you have

$\theta = \theta_0$.

Considering the two equations for incremental thrust coefficient, you get

$$4F\lambda^2 r\Delta r = \frac{\sigma c_l}{2} r^2 \Delta r.$$

Solving the above equation at each radial location r, the non-dimensional induced velocity (λ) at the point can be computed. You can use this to compute the thrust and torque coefficients.

Rotor Block

The Rotor block in Aerospace Blockset can be used to compute the aerodynamic forces and moments in all three dimensions for an isolated rotor or propeller. This computation requires the mechanical parameters of the rotor including the thrust and torque coefficients as inputs. These coefficients can be obtained experimentally by doing a static thrust and torque analysis. This is the approach typically followed, especially in case of small propellers used in multirotor vehicles like quadrotors. In scenarios where the experimental study is not possible, or a comparison is to be made between the experimentally obtained values and theoretical values, the Rotor block can be used.

The Rotor block assumes:

- Constant chord along the blade span
- Constant lift curve slope
- Profile drag contribution neglected
- Only linear, or ideal twist distribution

Variation of Thrust and Torque Coefficients with Collective Pitch Input

Compute the values of C_T and C_Q corresponding to different values of collective pitch input θ_0. Based on reference data, the range of collective pitch inputs considered here is from 0° to 12°.

```
theta0Array = 0:12;
```

The following matrices are used to save the thrust and torque coefficient results obtained using the two approaches. In case of BEMT computation, the torque coefficient values are computed with and without the inclusion of profile drag component. Save the values in the two columns of `CQArrayBEMT`.

```
CTArrayBEMT = zeros(length(theta0Array),1);
CQArrayBEMT = zeros(length(theta0Array),2);
CTArrayRotor = zeros(length(theta0Array),1);
CQArrayRotor = zeros(length(theta0Array),1);
```

Computation Using BEMT

Compute net thrust and torque coefficients using BEMT by calculating the incremental values of the coefficients at each radial location and then summing up.

```
Nelements = 100; % number of radial elements
xnodes = linspace((rootCutout-hingeOffset)/(radius-hingeOffset),1,Nelements+1); % radial location
delr = diff(xnodes); % corresponding to deltar in equations
```

Compute lift and drag coefficients, c_l and c_d at each radial location, using the function handles `'computeCl'` and `'computeCd'` with alpha (α) as the input argument.

```
computeCl=@(alpha)interp1(airfoilData.theta_NACA0015, airfoilData.CL_NACA0015, convang(alpha,'rad
computeCd=@(alpha)interp1(airfoilData.theta_NACA0015, airfoilData.CD_NACA0015, convang(alpha,'rad
```

Compute the angle of attack α at each radial location and convert to degrees using the `convang` function in Aerospace Toolbox. For interpolation, use `interp1` function in MATLAB®.

To incorporate tip loss, use Prandtl's tip-loss function. Implement this as a function handle with the non-dimensional radial location (r) and the inflow (λ) as input arguments.

```
F = @(r,lambda)(2/pi)*acos(exp(-.5*Nb*((1-r)/(r*(lambda/r)))));
```

To solve for λ at each radial location, use the `fsolve` (Optimization Toolbox) function in Optimization Toolbox. The default `'trust-region-dogleg'` algorithm is used here to solving the equation. Set the initial guess for the `fsolve` function using the `initInflow` variable.

```
initInflow = 0.01;
opts = optimset('Display','off');
```

Compute the net thrust and torque coefficients by looping over the range of collective pitch values, and the number of radial locations (for each pitch input).

Use the `convang` function to convert the collective pitch angle to radians. Compute the torque coefficient two ways, with and without the profile drag component.

```
for thetaInd = 1:length(theta0Array) % loop for collective pitch input
    theta0 = convang(theta0Array(thetaInd),'deg','rad');

    CT = 0; % initializing CT
    % initializing CQ
    CQwithDrag = 0; % with profile drag effect included
    CQwithoutDrag = 0; % without profile drag

    for rInd = 1:Nelements-1 % loop for radial locations
        r  =xnodes(rInd)+delr(rInd)/2;

        % solving for lambda at r
        lambdaVal = fsolve(@(lambda)4*F(r,lambda)*lambda^2*r-.5*sigma*r^2*(computeCl(theta0 - (la

        % computing incremental thrust coefficient
        dCT = 4*F(r,lambdaVal)*lambdaVal^2*r*delr(rInd);

        CT = CT + dCT;
        CQwithDrag = CQwithDrag + lambdaVal*dCT+ 0.5*sigma*computeCd(theta0 - (lambdaVal/r))*r^3*
        CQwithoutDrag = CQwithoutDrag + lambdaVal*dCT;
    end
    CTArrayBEMT(thetaInd) = CT;
    CQArrayBEMT(thetaInd,1) = CQwithDrag;
    CQArrayBEMT(thetaInd,2) = CQwithoutDrag;
end
```

Computation Using Rotor block

To compute the thrust and torque coefficients using Rotor block in the Aerospace blockset, use the Simulink® model, `CTCQcomputation.slx`.

The “Model Callbacks” function adds the reference parameters to the model and the mask parameters `R` (radius), `c` (chord), `e` (hinge offset), `clalpha` (lift curve slope) are set to the reference values. The twist type is set to linear with the root pitch angle (θ_0) set to the collective pitch angle (`theta0`). The input parameter to the block, density (ρ) i set to the approximate sea-level value of 1.224 kg/m^3, and rotor speed (Ω) is set using the reference data (rpm).

To loop over the varying collective pitch angles, use a `Simulink.SimulationInput` object.

```
model = 'CTCQcomputation';
open_system(model);

simin(1:length(theta0Array)) = Simulink.SimulationInput(model);

for i = 1:length(theta0Array)
    theta0 = convang(theta0Array(i),'deg','rad');
    simin(i) = simin(i).setVariable('theta0',theta0);
end

simout = sim(simin,'ShowProgress','off');% turning off simulation progress messages

for i = 1:length(theta0Array)
    CTArrayRotor(i) = simout(1,i).CT;
    CQArrayRotor(i) = simout(1,i).CQ;
end
```

Analyzing Results

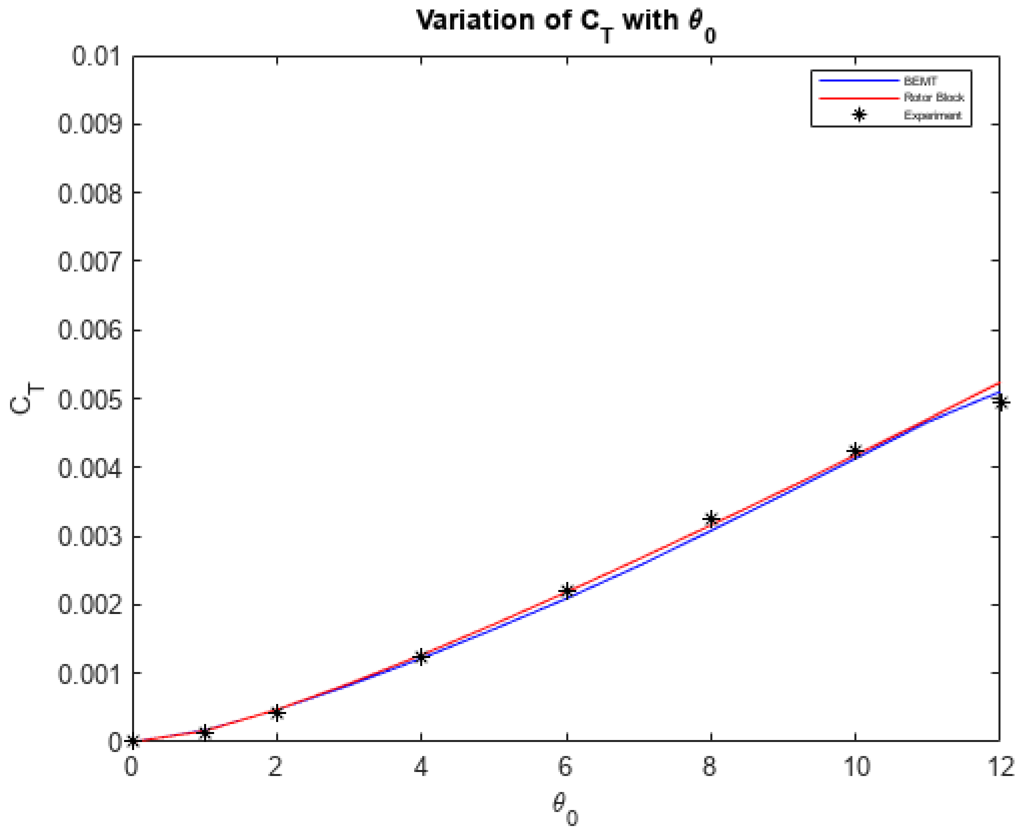
The experimental results for thrust and torque coefficients are obtained from [1] and saved in `Ref1Nb2Data.mat`. The experimental results present in the file are:

- `theta_data:` collective pitch angles at which data is available
- `CTdata`: thrust coefficient data
- `CQdata`: torque coefficient data

Variation of thrust coefficient with collective pitch

Plot the variation of C_T with θ_0.

```
figure,plot(theta0Array,CTArrayBEMT,'b',...
    theta0Array, CTArrayRotor, 'r',refData.theta_data, refData.CTdata, 'k*')
ylim([0 0.01]);
xlabel('\theta_0')
ylabel('C_T')
ax = gca;
ax.YRuler.Exponent = 0; % setting y-axis ticks to standard notation
title('Variation of C_T with \theta_0')
legend('BEMT','Rotor Block', 'Experiment',...
    'Location','best','NumColumns',1,'FontSize',5);
```

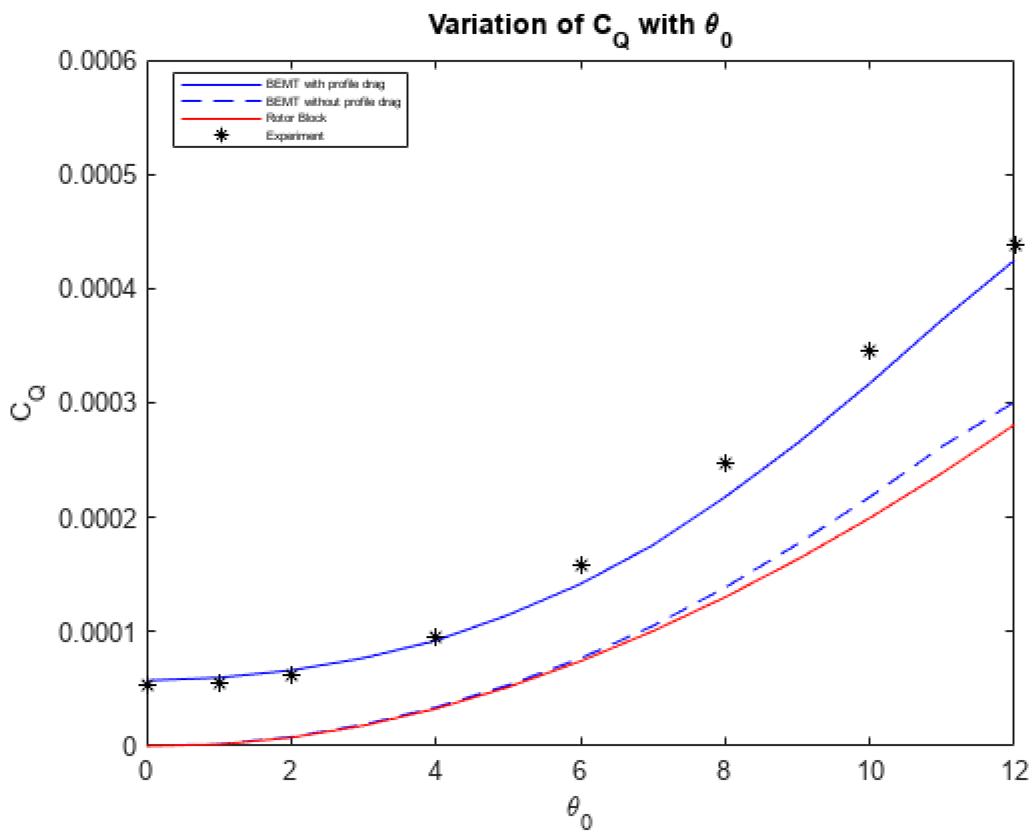


Good correlation is observed between the experimental results, the results obtained using BEMT and the computed values obtained using Rotor block. Even though the computation using Rotor block assumes constant value for the lift coefficient, the effect of varying lift coefficient along the span (with angle of attack) is minimal.

Variation of torque coefficient with collective pitch

Plot the variation of C_Q with θ_0.

```
figure,plot(theta0Array,CQArrayBEMT(:,1),'b',...
    theta0Array,CQArrayBEMT(:,2),'--b',theta0Array,CQArrayRotor,'r',...
    refData.theta_data, refData.CQdata, 'k*')
xlabel('\theta_0')
ylabel('C_Q')
ylim([0 0.0006]);
title('Variation of C_Q with \theta_0')
ax = gca;
ax.YRuler.Exponent = 0;
legend('BEMT with profile drag', 'BEMT without profile drag','Rotor Block', 'Experiment',...
    'Location','best','NumColumns',1,'FontSize',5);
```



Good correlation is observed between the results obtained using BEMT, without profile drag, and the results obtained using Rotor block. Similarly, good correlation can be seen between the results obtained using BEMT, with profile drag, and the experimental results.

In the torque coefficient computation using Rotor block, it is assumed that the drag coefficient is small, and hence the profile drag contribution is neglected.

The following analysis studies the contribution of profile drag component in torque computation. For small values of θ_0, the profile drag contribution over the entire rotor can be roughly approximated as

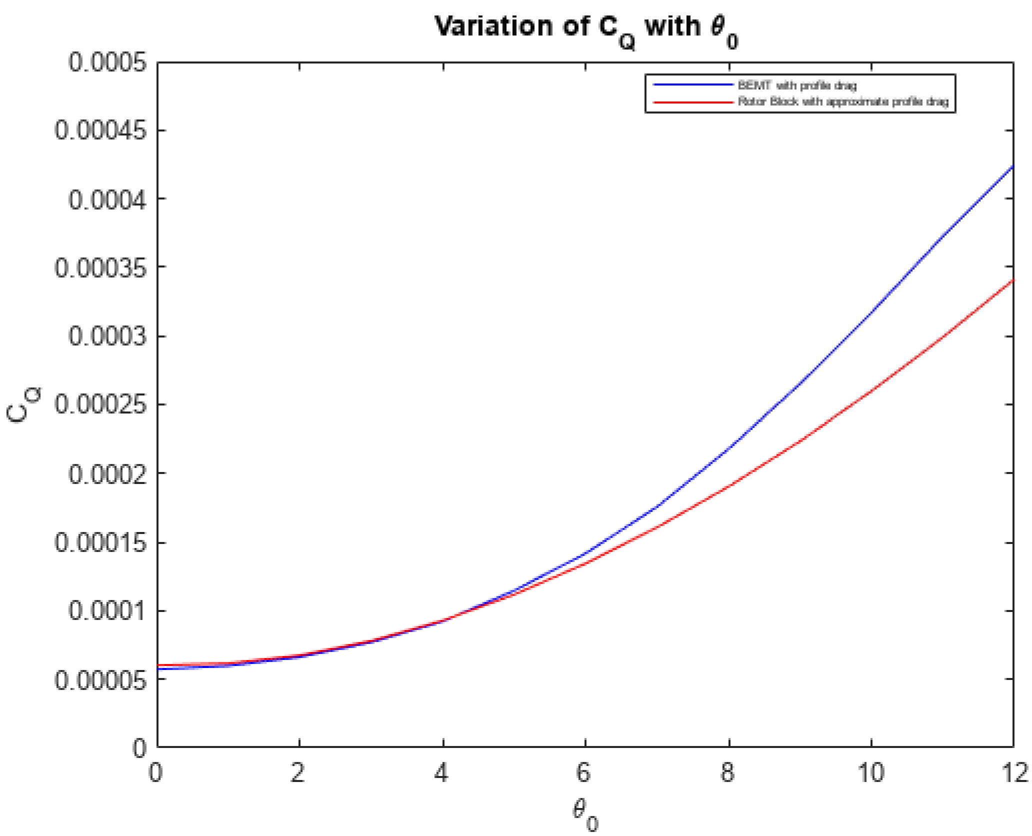
$$\int_0^1 \frac{\sigma c_d}{2} r^3 dr = \frac{\sigma c_{d0}}{8}.$$

Here, c_{d0} is considered as the mean drag coefficient.

Consider the approximate value of c_{d0} for NACA0015[2] in computing the profile drag component. Add the result to torque coefficient values obtained using Rotor block and compared with the BEMT values.

```
cd0 = 0.0113;
CQprofileApprox = sigma*cd0/8;
CQprofileRotor = CQArrayRotor+CQprofileApprox;
figure,plot(theta0Array,CQArrayBEMT(:,1),'b',theta0Array, CQprofileRotor,'r')
xlabel('\theta_0')
ylabel('C_Q')
ylim([0 0.0005]);
```

```
title('Variation of C_Q with \theta_0')
ax = gca;
ax.YRuler.Exponent = 0;
legend('BEMT with profile drag', 'Rotor Block with approximate profile drag',...
    'Location','best','NumColumns',1,'FontSize',5);
```



The analysis shows that for the specific rotor,

- the approximation works well for smaller pitch input values, as would be the case in case of smaller rotors or propellers.
- the effect of varying lift and drag coefficient values along the span on the computed torque coefficient is minimal.

Conclusion

The analysis shows that Rotor block can be used for the computation of thrust and torque coefficient values, with limitations.

- Constant chord: In case of rotors with varying chord, computing the mean geometric chord[3] and using it in the computation is a reasonable approximation.
- Constant lift coefficient: As shown in the above analysis, using the mean lift coefficient corresponding to the airfoil in the computation is a good approximation.
- Low profile drag: The block neglects the effects of profile drag in the computation of torque coefficient.

- Linear or ideal twist distribution: The block is enabled to consider linear or ideal twist variations along the span.

References

[1] Johnson W. (2013). Rotorcraft Aeromechanics. Cambridge University Press.

[2] Knight, M. and R. A. Hefner (1937). *Static thrust analysis of the lifting airscrew. Technical Report 626, National Advisory Committee on Aeronautics.*

[3] Leishman G.J. (2006). Principles of helicopter aerodynamics with CD extra. Cambridge University Press.

Fly the De Havilland Beaver with Unreal Engine Visualization

This example shows how to model the De Havilland Beaver using Simulink® and Aerospace Blockset™ software with Unreal Engine® (UE) visualization. It shows how to use a pilot joystick to fly the De Havilland Beaver in the Airport or Griffiss Airport scenes or the Cesium Ion® environment.

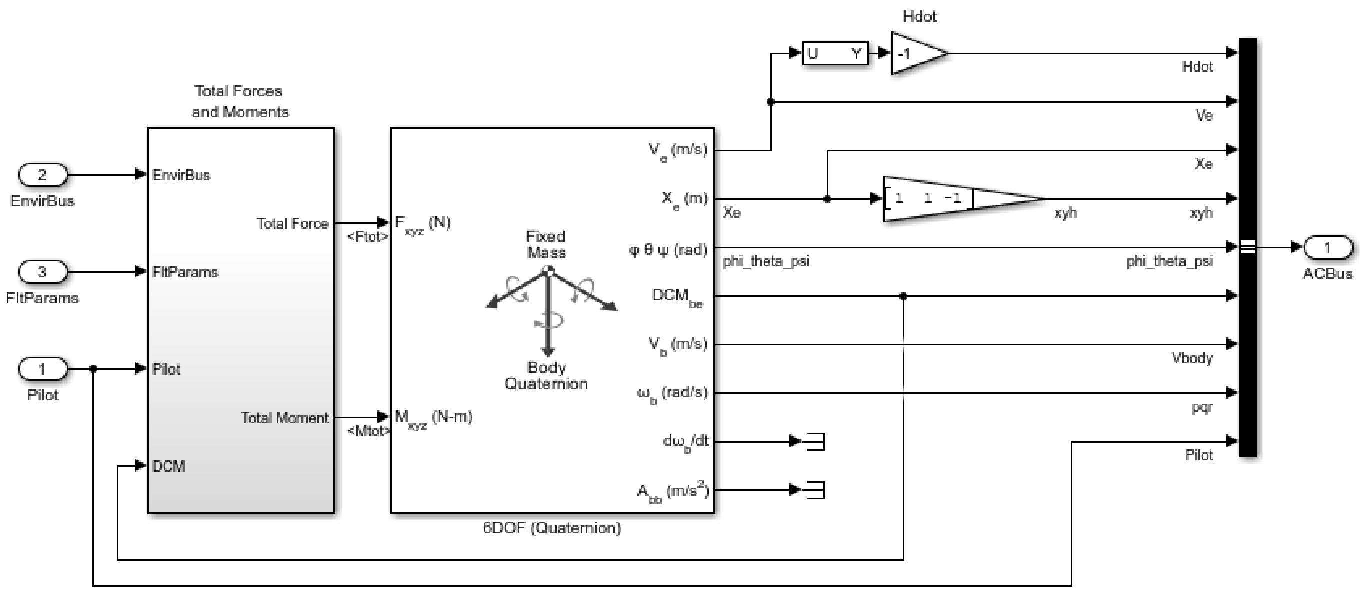
The De Havilland Beaver model includes airframe dynamics and aerodynamics. This also models effects of the atmosphere, such as wind profiles for the landing phase.

The “Fly the De Havilland Beaver” on page 9-14 example interfaces with the FlightGear flight simulator. This example explores the use of UE visualization.

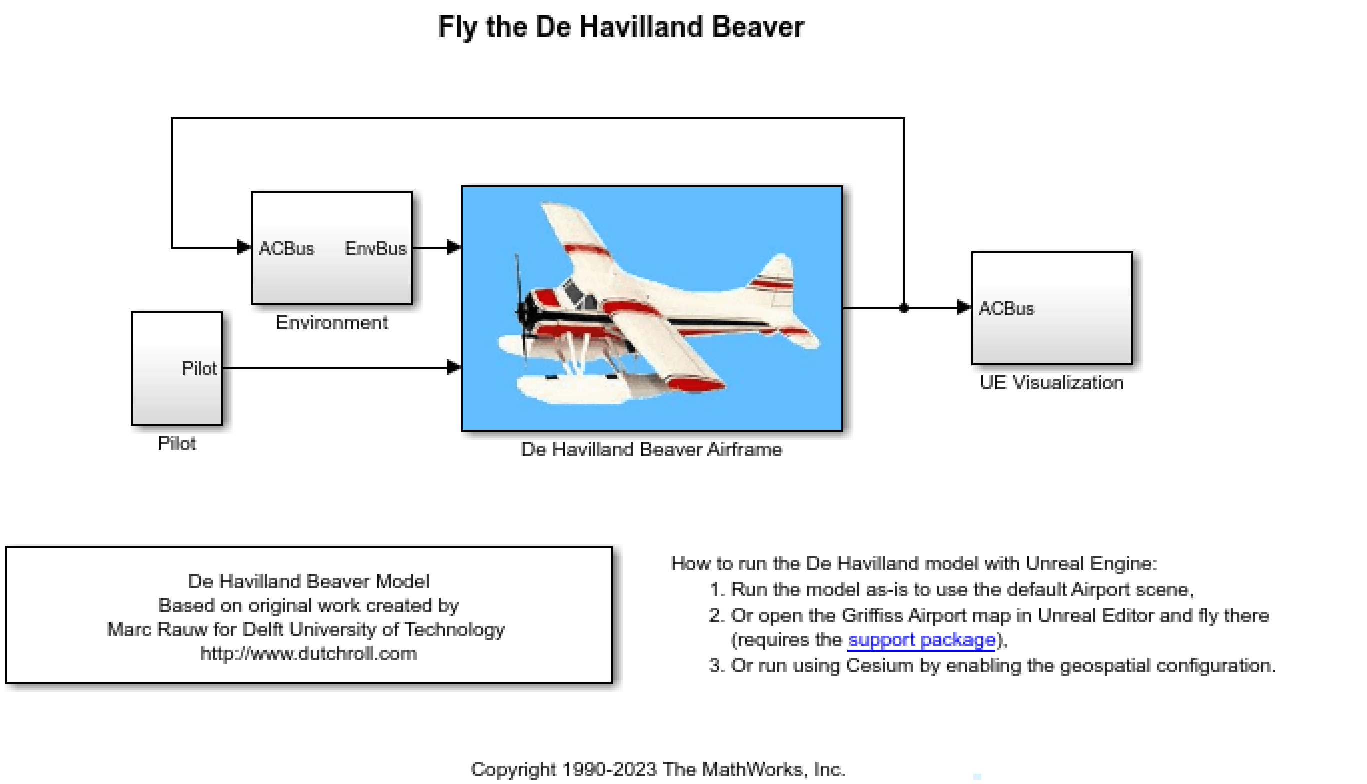
Note: This example is not supported in Simulink Online.

Explore UE Visualization

To begin the conversion, open the De Havilland Beaver Airframe > Aircraft Dynamics subsystem and add the pilot commands to `ACBus` to include the control surface movements in the visualization.



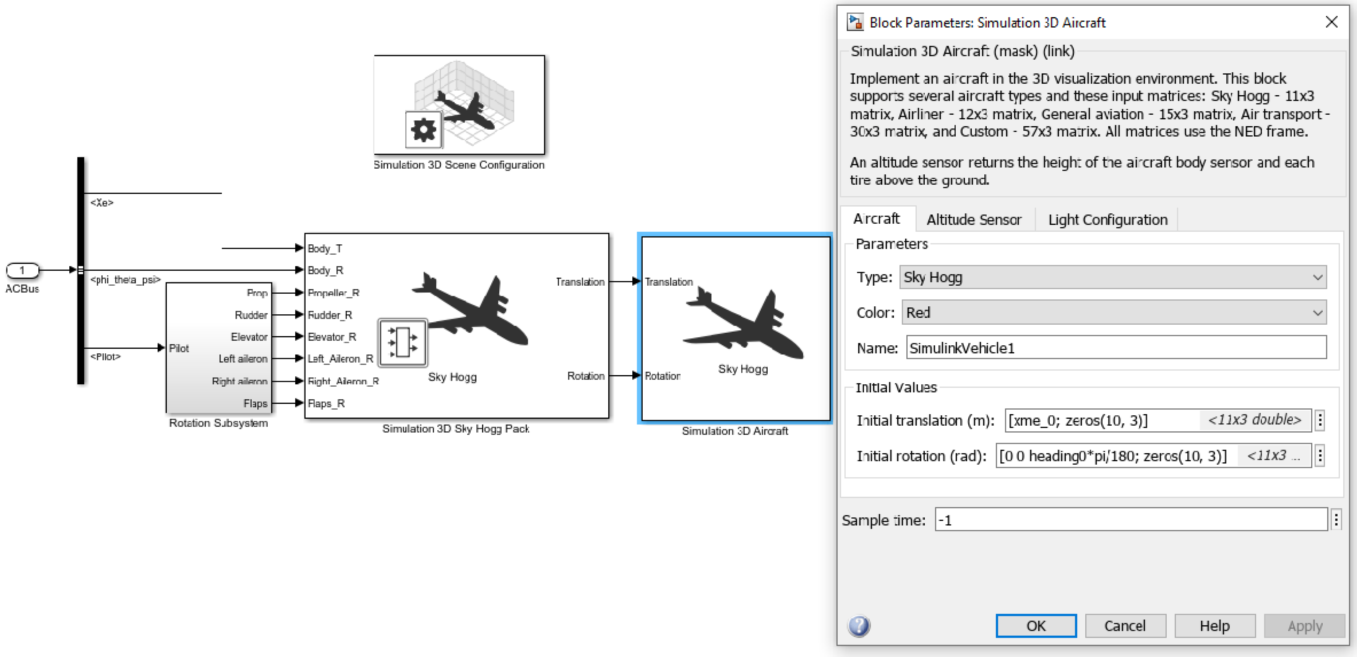
Replace the three animation and FlightGear blocks on the right side of the model with a single subsystem called "UE Visualization" that takes the `ACBus` as input.



In the new UE Visualization subsystem, add a Simulation 3D Aircraft block first, and then the Simulation 3D Scene Configuration block. Double-click the aircraft block and clear **Enable altitude and WoW sensors** in the **Altitude Sensor** tab, then click **OK**.

Add a Simulation 3D Sky Hogg Pack block and connect its output ports to Simulation 3D Aircraft. Open the pack block mask and select the **Propeller rotation** check box in the **Propulsion** tab. In the **Controls** tab press the **Select all** button, then click **OK**.

Connect the `ACBus` input to a bus selector and configure it to output `Xe` (body location), `[phi, theta, psi]` (body rotation), and `Pilot` (actuator commands). Send that data to the pack block and a rotation subsystem as shown.

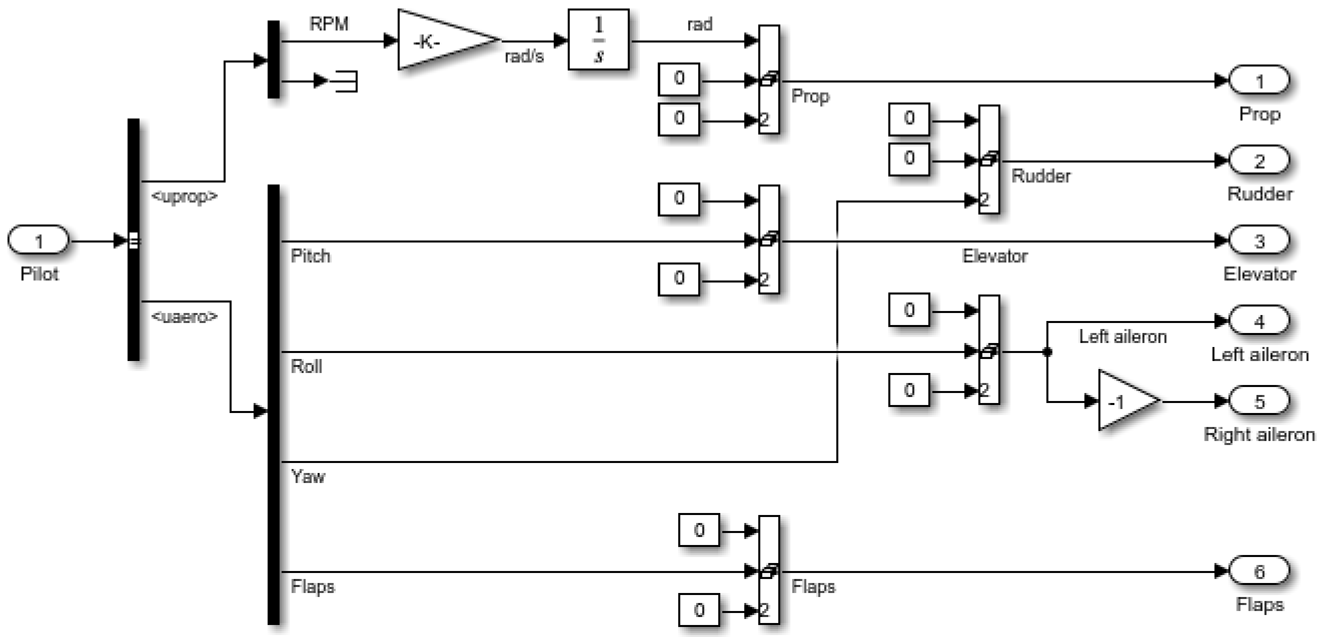


The **Sky Hogg** aircraft type represents the De Havilland Beaver, although it is a smaller and less massive airframe than the Beaver. To properly visualize the Beaver, create a skeletal mesh for it using the General Aviation skeleton and import that FBX file into Unreal Editor®. For more details, see "Prepare Custom Aircraft Mesh for the Unreal Editor" on page 4-34 and General Aviation.

The **Translation** and **Rotation** inputs to the Simulation 3D Aircraft block are sized as [11x3] as required for the **Sky Hogg** aircraft type. This table shows what part of the aircraft each of these affect. Note that, other than the body, only one rotation of the six degrees of freedom is enabled for each part.

Index:	Translation:	Rotation:	Sky Hogg Mesh Part:
1	X, Y, Z	Roll, Pitch, Yaw	BODY (Entire aircraft)
2	--	Roll	PROPELLER
3	--	Yaw	RUDDER
4	--	Pitch	ELEVATOR
5	--	Pitch	LEFT_AILERON
6	--	Pitch	RIGHT_AILERON
7	--	Pitch	FLAPS
8	--	Yaw	NOSE_WHEEL_STRUT
9	--	Pitch	NOSE_WHEEL
10	--	Pitch	LEFT_WHEEL
11	--	Pitch	RIGHT_WHEEL

The Rotation Subsystem creates [1x3] row vectors for each of the pilot actuator commands for the propeller RPM, rudder, elevator, left and right ailerons, and flaps. The RPM **Gain** block multiplies by 2*pi/60 to convert to radians per second.



Open the model.

```
mdl = "asbdhc2_FlyBeaverUE";
open_system(mdl);
```

Fly

Before running the model, note that **Simulation Pacing** has been turned on so that the simulation clock matches the wall clock.

- Make sure your joystick is connected.
- Click the **Run** button, then allow a few seconds for the 3D visualization window to initialize.
- Use the joystick to fly the aircraft.
- Once it is simulating, you can switch between camera views by first left-clicking inside the 3D window, then using the keys `0` through `9` to choose between ten preconfigured camera positions. For flight simulation, views `2` (behind) and `5` (cockpit) are the most useful. For more information on camera views, see the **Run Simulation** section in “Customize Scenes Using Simulink and Unreal Editor” on page 4-7.

Update to Griffiss Airport Using Custom Scenes and Fly

To fly in the Griffiss Airport (or a custom) scene:

1. Double-click the Simulation 3D Scene Configuration block to open its mask, and set the **Scene source** to **Unreal Editor**.
2. Enter the **Project** location, to which you saved the `AutoVrtlEnv.uproject` file from the support package, then click the **Open Unreal Editor** button.
3. Click **OK** to save your changes and close the mask.
4. Once Unreal Editor opens, change the map to Griffiss Airport by finding the folder MathWorksAerospaceContent Content > Maps and double-clicking `GriffissAirport`.

Change the values of the following workspace variables.

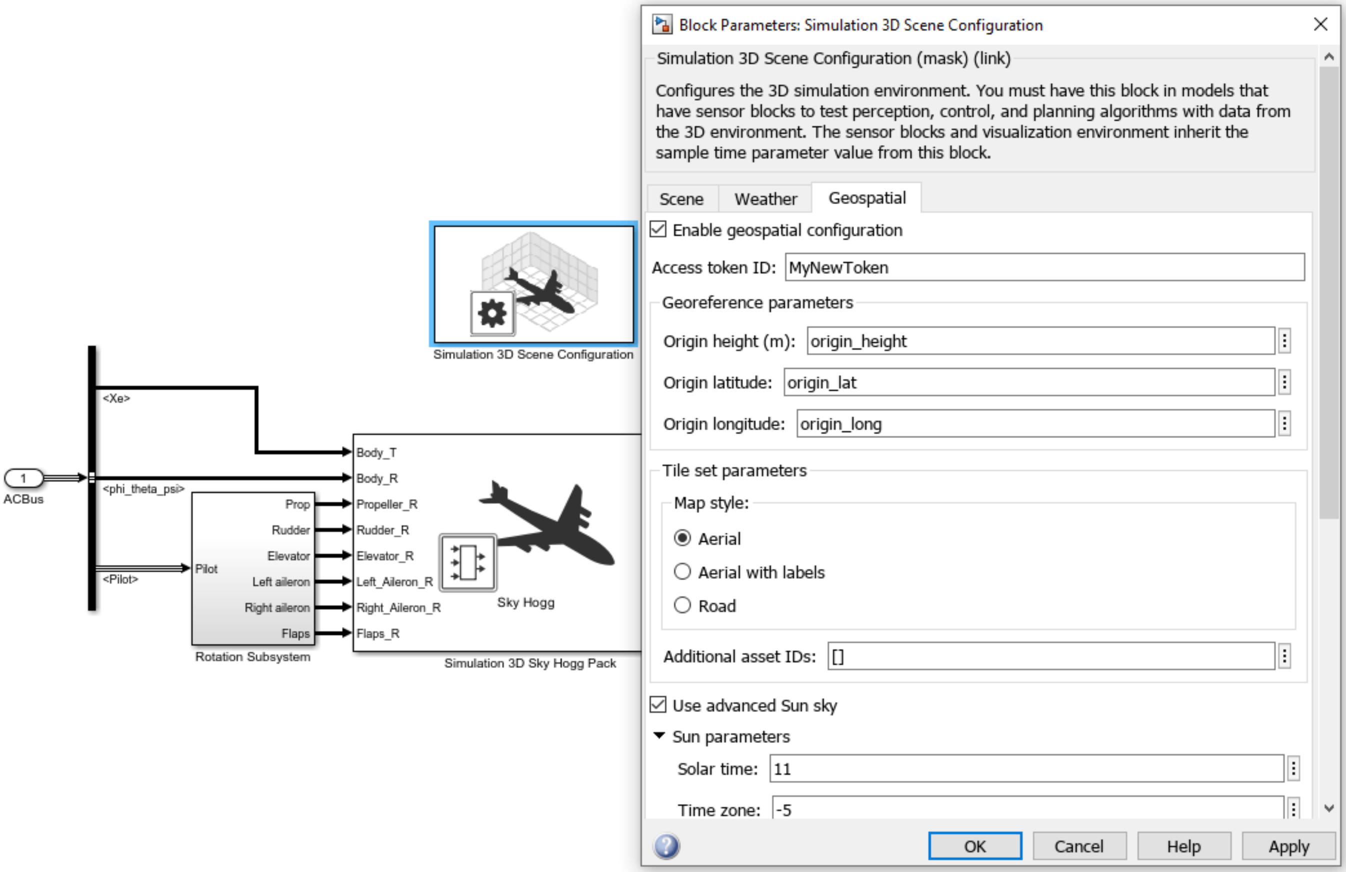
- `xme_0 = [0, 0, -200]` (place the aircraft 200 meters above the map origin)
- `heading0 = -45` (align with runway 33, which is about -45 degrees from true north)

Click the **Run** button. Once the model has compiled and "Initializing" displays on the bottom bar, click the **Play** button in Unreal Editor. Allow a few seconds for the connection to be made and simulation to begin.

Update to Boston Logan or Sedona Airport using Cesium and Fly

To fly with Cesium, which streams the 3D map and terrain data of a location:

1. First perform the “Visualize with Cesium” on page 2-42 if you have not already done so. The initial setup includes creating a Cesium Ion account and access token, and creating a new token in the Simulink **Authentication Manager** to hold your Cesium Ion token.
2. Open the Simulation 3D Scene Configuration block mask and select **Enable geospatial configuration** in the **Geospatial** tab.
3. Enter your Cesium Ion token name in **Access token ID**.
4. Select the **Use advanced Sun sky** option.
5. Click **Apply** to save your changes.
6. In the **Scene** tab, for **Scene source** choose either **Default Scenes** or **Unreal Editor**.
7. If using Unreal Editor for the Scene source, change the map to GeoSpatial by finding the folder MathWorksGeoSpatial Content > Maps and double-clicking `GeoSpatialMap`.
8. Click **OK** to save your changes and close the mask.



Note the workspace variables for the geospatial origin: `origin_height`, `origin_lat`, and `origin_long`. The initial heading is set by workspace variable `heading0` in the 6DOF (Quaternion) block shown in the first figure above.

Workspace Variable:	Description:
heading0	Initial heading in degrees from East
origin_height	Geospatial origin height in meters, which is the height above the WGS84 ellipsoid model of the Earth at the origin latitude and longitude
origin_lat	Geospatial origin latitude in degrees
origin_long	Geospatial origin longitude in degrees
xme_0	Initial [X,Y,Z] offset from the map origin; use [0,0,0] with Cesium

Change the values of the following workspace variables.

- `xme_0 = [0, 0, 0]` (no offset used with Cesium Ion)
- `heading0 = -135` (to align with Boston Logan airport runway 33, which is about -135 degrees from true east)

Click the **Run** button, then allow a few seconds for the 3D visualization window to initialize. Cesium Ion takes a few additional seconds for the 3D imagery to load.

The default geospatial origin is runway 33L at Boston Logan airport. To change the initial heading and location, change the values of the `heading0` and `origin_` variables listed above. For example, for the approach to runway 3 of Sedona Airport, use the following values.

- `heading0 = -40`
- `origin_height = 1500`
- `origin_lat = 34.841435`
- `origin_long = -111.797380`

See Also

Simulation 3D Aircraft | Simulation 3D Scene Configuration

External Websites

- Visualization Techniques with Aerospace Blockset Video

Aerospace Units Appendix

Aerospace Units

The main blocks of the Aerospace Blockset library support standard measurement systems. The Unit Conversion blocks support all units listed in this table.

Quantity	Metric (MKS)	English
Acceleration	meters/second^2 (m/s^2), kilometers/second^2 (km/s^2), (kilometers/hour)/second (km/h-s), g-unit (*g*)	inches/second^2 (in/s^2), feet/second^2 (ft/s^2), (miles/hour)/second (mph/s), g-unit (*g*)
Angle	radian (rad), degree (deg), revolution	radian (rad), degree (deg), revolution
Angular acceleration	radians/second^2 (rad/s^2), degrees/second^2 (deg/s^2), revolutions/minute (rpm), revolutions/second (rps)	radians/second^2 (rad/s^2), degrees/second^2 (deg/s^2), revolutions/minute (rpm), revolutions/second (rps)
Angular velocity	radians/second (rad/s), degrees/second (deg/s), revolutions/minute (rpm)	radians/second (rad/s), degrees/second (deg/s), revolutions/minute (rpm)
Density	kilogram/meter^3 (kg/m^3)	pound mass/foot^3 (lbm/ft^3), slug/foot^3 (slug/ft^3), pound mass/inch^3 (lbm/in^3)
Force	newton (N)	pound (lb)
Inertia	kilogram-meter^2 (kg-m^2)	slug-foot^2 (slug-ft^2)
Length	meter (m)	inch (in), foot (ft), mile (mi), nautical mile (nm, M)
Mass	kilogram (kg)	slug (slug), pound mass (lbm)
Pressure	Pascal (Pa)	pound/inch^2 (psi), pound/foot^2 (psf), atmosphere (atm)
Temperature	kelvin (K), degrees Celsius (^{o}C)	degrees Fahrenheit (^{o}F), degrees Rankine (^{o}R)
Torque	newton-meter (N-m)	pound-feet (lb-ft)
Velocity	meters/second (m/s), kilometers/second (km/s), kilometers/hour (km/h)	inches/second (in/s), feet/second (ft/s), feet/minute (ft/min), miles/hour (mph), knots