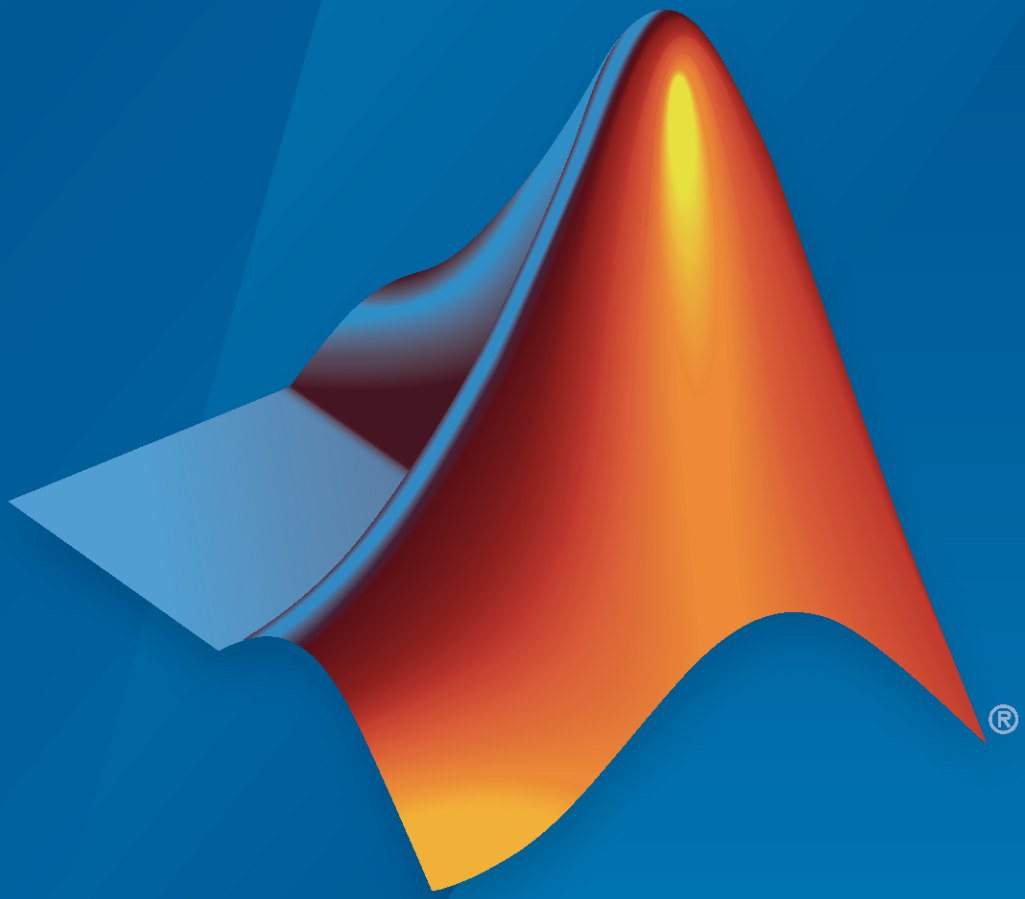
Mapping Toolbox™

User's Guide



MATLAB®

R2025b

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

Mapping Toolbox™ User's Guide

© COPYRIGHT 1997–2025 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

May 1997	First printing	New for Version 1.0
October 1998	Second printing	Version 1.1
November 2000	Third printing	Version 1.2 (Release 12)
July 2002	Online only	Revised for Version 1.3 (Release 13)
September 2003	Online only	Revised for Version 1.3.1 (Release 13SP1)
January 2004	Online only	Revised for Version 2.0 (Release 13SP1+)
April 2004	Online only	Revised for Version 2.0.1 (Release 13SP1+)
June 2004	Fourth printing	Revised for Version 2.0.2 (Release 14)
October 2004	Online only	Revised for Version 2.0.3 (Release 14SP1)
March 2005	Fifth printing	Revised for Version 2.1 (Release 14SP2)
August 2005	Sixth printing	Minor revision for Version 2.1
September 2005	Online only	Revised for Version 2.2 (Release 14SP3)
March 2006	Online only	Revised for Version 2.3 (Release 2006a)
September 2006	Seventh printing	Revised for Version 2.4 (Release 2006b)
March 2007	Online only	Revised for Version 2.5 (Release 2007a)
September 2007	Eighth printing	Revised for Version 2.6 (Release 2007b)
March 2008	Online only	Revised for Version 2.7 (Release 2008a)
October 2008	Online only	Revised for Version 2.7.1 (Release 2008b)
March 2009	Online only	Revised for Version 2.7.2 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.2 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 3.4 (Release 2011b)
March 2012	Online only	Revised for Version 3.5 (Release 2012a)
September 2012	Online only	Revised for Version 3.6 (Release 2012b)
March 2013	Online only	Revised for Version 3.7 (Release 2013a)
September 2013	Online only	Revised for Version 4.0 (Release 2013b)
March 2014	Online only	Revised for Version 4.0.1 (Release 2014a)
October 2014	Online only	Revised for Version 4.0.2 (Release 2014b)
March 2015	Online only	Revised for Version 4.1 (Release 2015a)
September 2015	Online only	Revised for Version 4.2 (Release 2015b)
March 2016	Online only	Revised for Version 4.3 (Release 2016a)
September 2016	Online only	Revised for Version 4.4 (Release 2016b)
March 2017	Online only	Revised for Version 4.5 (Release 2017a)
September 2017	Online only	Revised for Version 4.5.1 (Release 2017b)
March 2018	Online only	Revised for Version 4.6 (Release 2018a)
September 2018	Online only	Revised for Version 4.7 (Release 2018b)
March 2019	Online only	Revised for Version 4.8 (Release 2019a)
September 2019	Online only	Revised for Version 4.9 (Release 2019b)
March 2020	Online only	Revised for Version 4.10 (Release 2020a)
September 2020	Online only	Revised for Version 5.0 (Release 2020b)
March 2021	Online only	Revised for Version 5.1 (Release 2021a)
September 2021	Online only	Revised for Version 5.2 (Release 2021b)
March 2022	Online only	Revised for Version 5.3 (Release 2022a)
September 2022	Online only	Revised for Version 5.4 (Release 2022b)
March 2023	Online only	Revised for Version 5.5 (Release 2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)
September 2025	Online only	Revised for Version 25.2 (R2025b)

Contents

1 Get Started

Mapping Toolbox Product Description 1-2

Acknowledgments 1-3

Create Your First World Map 1-4

What Is a Map? 1-11

What Is Geospatial Data? 1-12

Vector Geodata 1-13

Raster Geodata 1-14
Digital Elevation Data 1-14
Remotely Sensed Image Data 1-14

Latitude and Longitude 1-15
Plot Latitude and Longitude 1-15

2 Data Import and Export

Supported Geospatial File Formats for Import and Export 2-2
Supported Import Formats 2-2
Supported Export Formats 2-3
Tips 2-3

Mapping Toolbox Data Sets 2-4
World Data 2-4
United States Data 2-6

Find Geospatial Vector Data 2-8

Find Geospatial Raster Data 2-9
Download Data 2-9
Use Web Map Service Data 2-10

Functions That Read and Write Geospatial Data 2-11

Data Stored in OpenStreetMap Layers **2-13**
Data Layers **2-13**
Building Layers **2-17**

Read Data from OpenStreetMap Files **2-21**

Display Buildings from OpenStreetMap Files **2-27**

Export Vector Geodata **2-33**

Export KML Files for Viewing in Earth Browsers **2-34**
Generate a Single Placemark Using kmlwritepoint **2-34**
Generate Placemarks from Addresses **2-35**
Export Point Geostructs to Placemarks **2-35**

Select Shapefile Data to Read **2-39**
Example 1: Predicate Function in Separate File **2-39**
Example 2: Predicate as Function Handle **2-40**
Example 3: Predicate as Anonymous Function **2-40**
Example 4: Predicate (Anonymous Function) Defined Within Cell Array . **2-41**
Example 5: Parametrizing the Selector; Predicate as Nested Function . . **2-41**

Exporting Vector Data to KML **2-43**

Export Images and Raster Grids to GeoTIFF **2-53**

Create, Process, and Export Digital Surface Model from Lidar Data . . . **2-66**

Visualize Sea Ice Concentrations from GRIB Data **2-74**

Convert Coastline Data (GSHHG) to Shapefile Format **2-82**

3 Map Display

Choose a 2-D Map Display **3-2**

Create Common Plots Using Map Axes **3-6**

Change Projection and Projection Parameters **3-12**

Create Map Axes in App **3-17**
Create New App **3-17**
Create App Components **3-17**
Set Up Map **3-17**
Program Component Behavior **3-19**
Run the App **3-20**
Example: App That Contains Map Axes **3-20**

axesm-Based Maps **3-22**
Tips to Working with axesm-Based Maps **3-23**

Create Maps of World and US Regions **3-24**
Create Maps of World Regions **3-24**
Create Maps of US Regions **3-25**

Switch Between Projections **3-29**
Change Projection Updating Meridian and Parallel Labels **3-29**
Change Projection Resetting Frame Limits **3-31**

Reprojection of Graphics Objects **3-34**
Auto-Reprojection of Mapped Objects and Its Limitations **3-34**
Reprojectability of Maps Generated Using geoshow **3-35**

Change Map Projections When Using geoshow **3-37**
Change Map Projections When Using geoshow **3-37**

Display Vector Data as Points and Lines **3-41**

Display Vector Data as Lines and Patches **3-46**

Create Maps Using Icons **3-50**
Represent Point Data **3-50**
Display Quantitative Data at Representative Points **3-51**
Animate Movement Along Path **3-53**
Annotate Basemap **3-54**

Control Color and Transparency of Icon Chart **3-56**
Remove Background from Custom Icon **3-56**
Change Color of Pushpin Icon **3-58**

Summary of Polygon Display Functions **3-62**
geoplot **3-62**
geoshow and mapshow **3-62**
patchm, patchesm, fillm, and fill3m **3-62**

Types of Data Grids and Raster Display Functions **3-64**

Fit Gridded Data to the Graticule **3-65**
Fit Gridded Data to Fine and Coarse Graticules **3-65**

Create 3-D Displays with Raster Data **3-68**
Display Elevation Data in 3-D **3-68**

Create Map Displays with Geographic Data **3-71**

Create Maps with Data in Projected Coordinate Reference Systems . . . **3-81**

Interactively Identify Geographic Locations **3-89**

Interactively Display Text on Maps **3-93**

Add Data Tips to Point, Line, and Polygon Shapes **3-97**
Programmatically Add Data Tips **3-97**
Interactively Add Data Tips **3-103**
Tips **3-105**

Work with Objects by Name **3-107**
Manipulate Displayed Map Objects By Name **3-107**

Create an Interactive Map for Selecting Point Features **3-112**

4 Map Customization

Add Labels and Annotations to Map Axes **4-2**

Customize Appearance of Map Axes **4-10**

Create Map of Quadrangle Using Cartographic Map Layout **4-16**

The Map Frame **4-23**

Map and Frame Limits **4-25**

Map Limit Properties **4-26**
Specify Map Projection Origin and Frame Limits Automatically **4-27**
Create Cylindrical Projection Using Map Limit Properties **4-30**
Create Conic Projection Using Map Limit Properties **4-31**
Create Southern Hemisphere Conic Projection **4-32**
Create North-Polar Azimuthal Projection **4-33**
Create South-Polar Azimuthal Projection **4-34**
Create Equatorial Azimuthal Projection **4-35**
Create General Azimuthal Projection **4-36**
Create Long Narrow Oblique Mercator Projection **4-37**

The Map Grid **4-39**
Control Grid Spacing **4-39**
Layer Grids **4-39**
Limit Grid Lines **4-39**
Label Grids **4-40**

Inset Maps **4-42**

Customize Appearance of Vector Basemaps **4-45**

Create Common Plots over Basemap Images **4-52**

Change Projection of Basemap Image **4-59**

Use Basemaps in Offline Environments **4-64**
Temporary Internet Access **4-65**
No Internet Access **4-68**

Create Base Layer from OpenStreetMap Data **4-76**

Display Data over Basemaps When Internet Access Is Intermittent **4-79**

Thematic Maps **4-82**
Choropleth and Classification Maps **4-82**
Scatter and Bubble Maps **4-82**
Contour Maps **4-83**
Stem Maps **4-83**

Create Choropleth Map of Population Density **4-85**

Create Choropleth Map from Table Data **4-88**

Create Classification Map from Table Data **4-92**

5 3-D Visualization

Access Basemaps and Terrain for Geographic Globe **5-2**
Use Installed Basemap **5-2**
Download Basemaps **5-2**
Add Custom Basemaps **5-2**
Access Terrain **5-3**
Specify Basemaps and Terrain **5-3**

Create Interactive Basemap Picker **5-4**

Visualize Aircraft Line-of-Sight over Terrain **5-6**

Visualize UAV Flight Path on 2-D and 3-D Maps **5-15**

Add Light Source to Terrain Map **5-25**

Surface Relief Shading **5-28**
Create Monochrome Shaded Relief Map **5-28**

Colored Surface Shaded Relief **5-32**
Create Colored Shaded Relief Map **5-32**

Relief Mapping with Light Objects **5-35**
Illuminate Color 3-D Relief Maps with Light Objects **5-35**

Drape Data on Elevation Maps **5-39**
Combine Elevation Maps with Other Kinds of Data **5-39**
Drape Data over Terrain with Different Gridding **5-39**

Drape Geoid Heights over Topography **5-44**

6 Data Analysis

Create Geospatial Tables **6-2**
Read Geospatial Table from File **6-2**
Create Geospatial Table from Table **6-4**
Create Geospatial Table from Arrays **6-5**
Tips **6-6**

Geographic Data Structures **6-7**
Fields of Geographic Data Structures **6-7**
Read Geographic Data Structures from Shapefiles **6-8**
Create Geographic Data Structures **6-9**

Combine Vector Data Using Table Joins **6-14**

Create and Display Polygons **6-17**
Simple Polygon **6-17**
Polygons with Holes or Multiple Regions **6-18**
Polygons Using Geographic Coordinates **6-20**
Filled Region of Polygons Using Geographic Coordinates **6-22**

Compare Polygons and Line Segments **6-25**

Simplify Vector Coordinate Data **6-27**

Geographic Interpolation of Vectors **6-28**

Find Vector Intersection Points **6-30**

Geographic Statistics for Point Locations on a Sphere **6-32**
Geographic Means **6-32**
Geographic Standard Deviation **6-33**

Equal-Areas in Geographic Statistics **6-36**
Geographic Histograms **6-36**
Converting to an Equal-Area Coordinate System **6-37**

Georeferenced Raster Data **6-39**
Reference Objects **6-39**
Referencing Matrices **6-39**
Referencing Vectors **6-39**

Spatially Reference Imported Rasters **6-41**
Differentiate Between Cells and Postings **6-41**
Spatially Reference an Image **6-42**
Spatially Reference an Elevation Grid **6-43**

Merge Multiple Raster Tiles Using Datastore **6-45**
Create Custom Datastore **6-45**
Merge DTED Tiles Using Datastore **6-48**
Adapt DTED Datastore for Other Data Sets **6-50**

Construct a Global Data Grid **6-51**

Geolocated Data Grids **6-53**
Define Geolocated Data Grid **6-53**

Geographic Interpretations of Geolocated Grids **6-56**
Type 1: Values Associated with the Upper Left Grid Coordinate **6-56**
Type 2: Values Centered Within Four Adjacent Coordinates **6-57**
Ordering of Cells **6-58**
Transform Regular to Geolocated Grids **6-58**
Transforming Geolocated to Regular Grids **6-58**

Data Grids as Logical Variables **6-59**

Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid **6-60**

Georeference an Image to an Orthotile Base Layer **6-63**

Unproject a Digital Elevation Model (DEM) **6-73**

Define Areas of Interest **6-82**

Clip Vector Data to Area of Interest **6-91**

Crop Raster Data to Area of Interest **6-95**

Query Points in Area of Interest **6-102**

Convert Vector Data to Raster Format **6-107**
Creating Data Grids from Vector Data **6-107**

Combine Vector and Raster Geodata on the Same Map **6-111**
Combining Raster Data and Vector Data on the Same Map **6-111**

Determine and Visualize Visibility Across Terrain **6-113**
Compute Line of Sight **6-113**

7 Web Map Service and Web Maps

Basic WMS Terminology **7-2**

Basic Workflow for Creating WMS Maps **7-3**

Create WMS Maps When Internet Access Is Intermittent **7-6**

Explore Layers on Same Server **7-9**

Search for Layers Outside the WMS Database **7-12**

Overlay Multiple WMS Layers **7-14**
Create Composite Map of Multiple Layers from One Server **7-14**

Combine Layers from One Server with Data from Other Sources 7-15
Drape Orthoimagery Over DEM 7-17

Add Legend to WMS Map 7-20

Animate Data from WMS Layers 7-24
Create Movie of Terra/MODIS Maps 7-24
Create Animated GIF of WMS Maps 7-26
Create Animation of Time-Lapse Radar Observations 7-27

Display Animation of Radar Images over GOES Backdrop 7-30

Add Custom Basemaps from WMS Data 7-32

Modify Your WMS Map Request 7-36
Set Map Request Geographic Limits and Time 7-36
Manually Edit Web Map Request URL 7-37

Read WMS Maps Using Different Coordinate Reference Systems 7-39

Read Quantitative Data from WMS Server 7-43
Read and Display Terrain Elevation Data 7-43
Merge Elevation Data with Rasterized Vector Data 7-44
Drape Basemap Imagery onto Elevation Data 7-47
Tips 7-50

Write WMS Layers to KML File 7-52

Troubleshoot WMS Servers 7-54
Connection Errors 7-54
Intermittent Internet Access 7-55
Geographic Limits in Descending Order or Out of Bounds 7-55
Non-EPSG:4326 Coordinate Reference Systems 7-55
Map Not Returned 7-56
Unsupported WMS Version 7-57
Other Unrecoverable Server Errors 7-57

Composite and Animate Web Map Service (WMS) Meteorological Layers 7-58

Web Map Migration Strategies 7-70
Benefits of Geographic Axes 7-70
Replacement Functions 7-70
Update Code to Use Geographic Axes 7-71
Interact with Geographic Axes 7-73

Use Base Layers in Geographic Axes 7-75
Benefits of Geographic Axes 7-75
Replace Base Layers 7-75
Interactively Select Basemaps 7-78

Add Interactive Controls to Geographic Axes 7-80
Add Basemap Picker to Geographic Axes 7-80
Add Zoom Slider to Geographic Axes 7-82
Get Coordinates of Pointer 7-84

8 Geometric Geodesy

Shape of the Earth **8-2**
Ellipsoid Shape **8-2**
Geoid Shape **8-2**

Comparison of Reference Spheroids **8-4**
referenceSphere Objects **8-4**
referenceEllipsoid Objects **8-6**
World Geodetic System 1984 . . . . . **8-8**
Ellipsoid Vectors **8-9**
oblateSpheroid Objects **8-10**

Work with Reference Spheroids **8-11**
Map Projections **8-11**
Curves and Areas **8-12**
3-D Coordinate Transformations **8-12**

Comparison of 3-D Coordinate Systems **8-13**
Earth-Centered Earth-Fixed Coordinates **8-13**
Geodetic Coordinates **8-14**
East-North-Up Coordinates **8-15**
North-East-Down Coordinates **8-15**
Azimuth-Elevation-Range Coordinates **8-16**
Tips **8-17**

Vectors in 3-D Coordinate Systems **8-18**
Tips **8-19**

Plotting a 3-D Dome as a Mesh over a Globe **8-21**

Find Ellipsoidal Height from Orthometric Height **8-27**
Find Ellipsoidal Height from Orthometric and Geoid Height **8-28**

Angle Representations and Angular Units **8-30**
Degrees and Radians **8-30**
Degree Representations **8-30**
Latitude and Longitude Formats **8-31**

Length and Distance Units **8-33**
Choosing Units of Length **8-33**
Converting Units of Length **8-33**

Distances on the Sphere **8-35**
Arc Length as an Angle in the distance and reckon Functions **8-36**
Summary: Available Distance and Angle Conversion Functions **8-36**

Comparison of Rhumb Lines and Great Circles **8-38**
Rhumb Lines **8-38**
Great Circles **8-39**
Use Rhumb Lines and Great Circles in Functions **8-39**

Generate Small Circles **8-40**

Interactively Create Small Circle and Track Annotations on Maps **8-42**

9 Coordinate Reference Systems

Summary and Guide to Projections **9-2**
Cylindrical Projections **9-2**
Pseudocylindrical Projections **9-5**
Conic Projections **9-9**
Pseudoconic Projections **9-10**
Polyconic Projections **9-11**
Azimuthal Projections **9-12**
Pseudoazimuthal Projections **9-14**
Modified Azimuthal Projections **9-14**

The Three Main Families of Map Projections **9-15**
Unwrapping the Sphere to a Plane **9-15**
Cylindrical Projections **9-15**
Conic Projections **9-16**
Azimuthal Projections **9-17**

Quantitative Properties of Map Projections **9-19**
Shape **9-19**
Distance **9-19**
Direction **9-19**
Scale **9-19**
Area **9-19**

Projection Parameters **9-21**
Projection Characteristics Maps Can Have **9-21**

Transform Coordinates to a Different Projected CRS **9-27**

Project and Display Raster Data **9-30**
Project Raster Data **9-30**
Unproject Raster Data **9-31**

Create UTM Maps **9-33**
Create UTM Map Using Map Axes **9-34**
Create UTM Map Using axesm-Based Maps **9-35**

Set UTM Parameters Interactively **9-37**

Work in UTM Without a Displayed Map **9-40**

Use the Transverse Aspect to Map Across UTM Zones **9-42**

Map Projections and Distortions **9-44**
Use Inverse Projection to Recover Geographic Coordinates **9-44**
Projection Distortions **9-44**

Visualize Spatial Error Using Tissot Indicatrices **9-46**
Visualize Projection Distortions Using Tissot Indicatrices **9-46**

Quantify Map Distortions at Point Locations **9-49**
Use distortcalc to Determine Map Projection Geometric Distortions **9-49**

Projection Aspect **9-53**
The Orientation Vector **9-53**
Control the Map Projection Aspect with an Orientation Vector **9-55**

Rotational Transformations on the Globe **9-59**
Reorient Vector Data with rotatem **9-59**
Reorient Gridded Data **9-61**

10 Map Projections — Alphabetical List

11 Applications

Build Pikes Peak RoadRunner 3D Scene Using RoadRunner HD Map **11-2**

Visualize Viewsheds and Coverage Maps Using Terrain **11-9**

Visualize Coverage Maps over Lunar Terrain Using Ray Tracing **11-40**

Radar Performance Analysis over Terrain **11-58**

Radar Vertical Coverage over Terrain **11-68**

Process Large GeoTIFF File as Blocked Image **11-77**

Crop and Mask Large GeoTIFF File Using Shapefile **11-85**

Hurricane Natural Catastrophe Risk Estimation **11-92**

Get Started

- “Mapping Toolbox Product Description” on page 1-2
- “Acknowledgments” on page 1-3
- “Create Your First World Map” on page 1-4
- “What Is a Map?” on page 1-11
- “What Is Geospatial Data?” on page 1-12
- “Vector Geodata” on page 1-13
- “Raster Geodata” on page 1-14
- “Latitude and Longitude” on page 1-15

Mapping Toolbox Product Description

Analyze and visualize geographic information

Mapping Toolbox provides algorithms and functions for transforming geographic data and creating map displays. You can visualize your data in a geographic context, build map displays from more than 60 map projections, and transform data from a variety of sources into a consistent geographic coordinate system.

Mapping Toolbox supports a complete workflow for managing geographic data. You can import vector and raster data from a wide range of file formats and web map servers. The toolbox lets you process and customize data using trimming, interpolation, resampling, coordinate transformations, and other techniques. Data can be combined with base map layers from multiple sources in a single map display. You can export data in file formats such as shapefile, GeoTIFF, and KML.

Acknowledgments

This software was originally developed and maintained through Version 1.3 by Systems Planning and Analysis, Inc. (SPA), of Alexandria, Virginia.

Except where noted, the information contained in example and sample data files (found in *matlabroot*`/examples/map/data` and *matlabroot*`/toolbox/map/mapdata`) is derived from publicly available digital data sets. These data files are provided as a convenience to Mapping Toolbox users. MathWorks® makes no claims that any of this data is free of defects or errors, or that the representations of geographic features or names are up to date or authoritative.

Create Your First World Map

Geospatial data can be complex and difficult to process. Mapping Toolbox™ functions handle many of the details of loading, representing, and displaying spatial data. *Spatial data* describes location, shape, and spatial relationships. *Geospatial data* is spatial data that is georeferenced, or tied to, specific locations on, under, or above the surface of a planet.

This page shows how to create similar world maps using map axes (*since R2023a*) and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Load Data

Load data to use in the examples.

Read three shapefiles into the workspace by using the `readgeotable` function.

- A shapefile containing world land areas. The file represents the land areas using polygons in geographic coordinates.
- A shapefile containing world rivers. The file represents the rivers using lines in geographic coordinates.
- A shapefile containing world cities. The file represents the cities using points in geographic coordinates.

```
land = readgeotable("landareas.shp");
rivers = readgeotable("worldrivers.shp");
cities = readgeotable("worldcities.shp");
```

Load a MAT file containing the coordinates of global coastlines. The variables within the MAT file, `coastlat` and `coastlon`, specify numeric latitude and longitude coordinates, respectively.

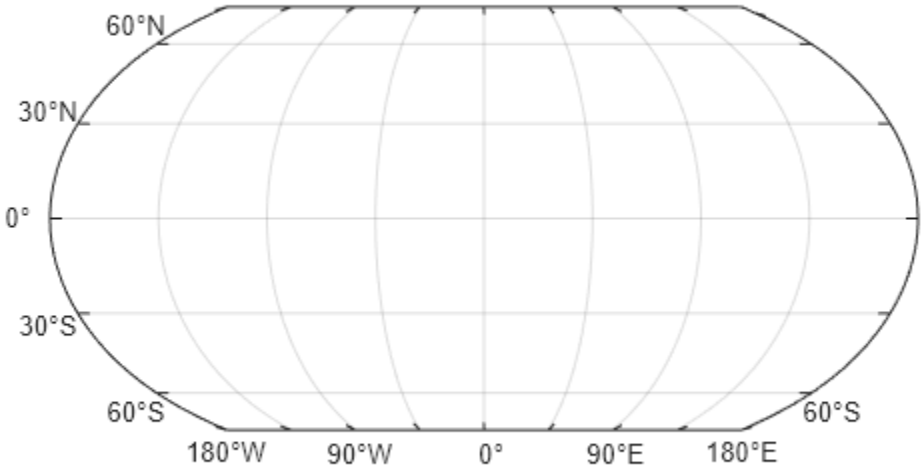
```
load coastlines
```

Create Map Using Map Axes

Create a world map using a map axes object.

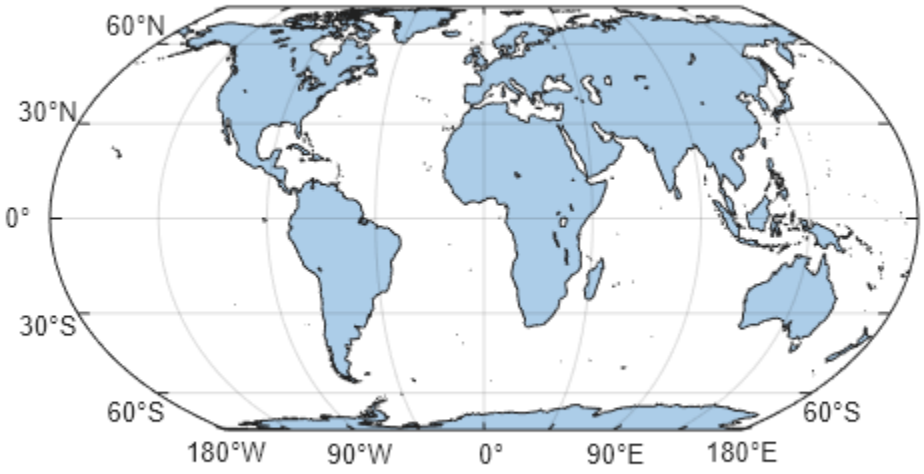
Set up a new map by using the `newmap` function. By default, map axes objects use an Equal Earth projection centered on the prime meridian and the equator.

```
figure
newmap
```



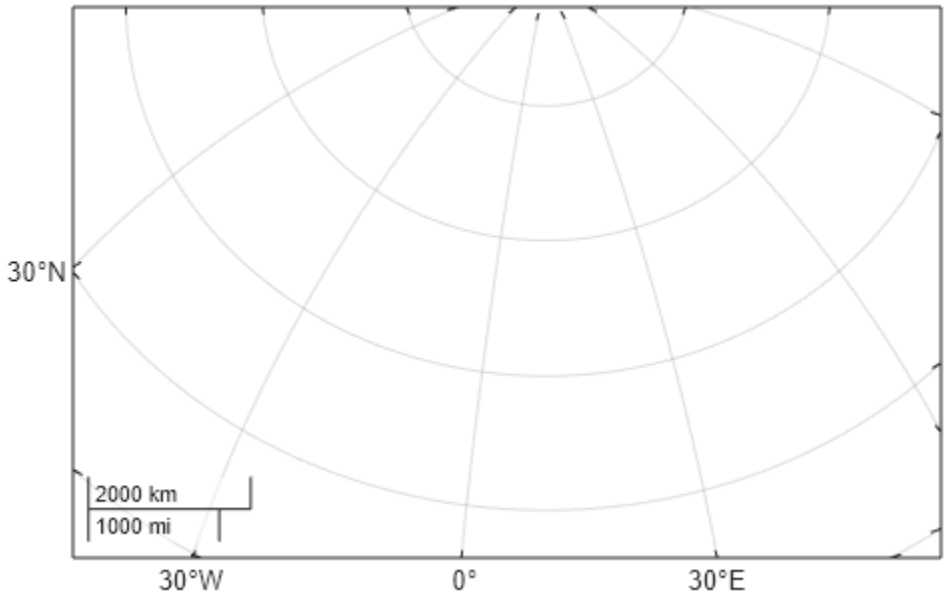
Display the land areas using the `geoplot` function.

```
geoplot(land)
```



You can use map axes to create maps in any supported projected coordinate reference system (CRS). Set up a new map using a projected CRS that is appropriate for Europe. Create the CRS using the `projcrs` function and the EPSG code `3035`, which uses a Lambert Azimuthal Equal Area projection method.

```
figure
p = projcrs(3035);
newmap(p)
```



Display the land areas as green polygons, the rivers as blue lines, and the cities as red points.

```
geoplot(land,FaceColor=[0.65 0.85 0.45])
hold on
geoplot(rivers,Color=[0 0.4470 0.7410])
geoplot(cities,MarkerEdgeColor=[0.6350 0.0780 0.1840])
```

Label the Atlantic Ocean. Specify the location of the text using latitude and longitude coordinates.

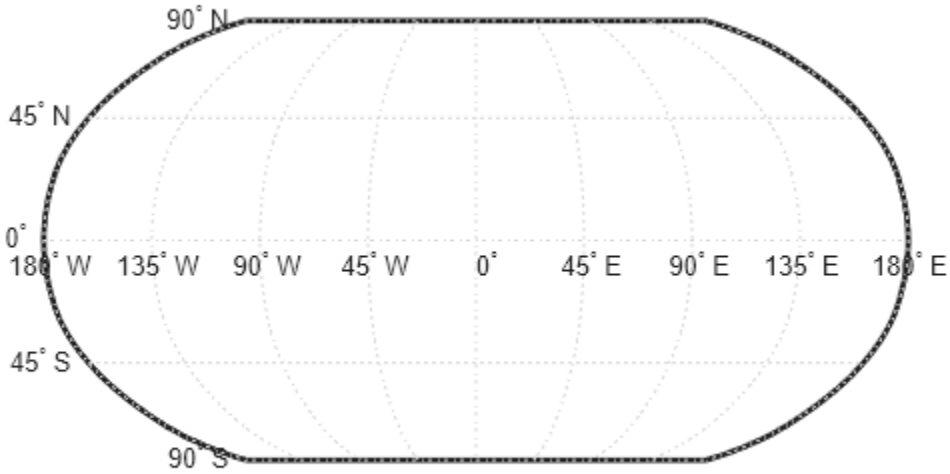
```
text(39,-43,"Atlantic Ocean")
```

Create Map Using `axesm`-Based Map

Create a world map using an `axesm`-based map.

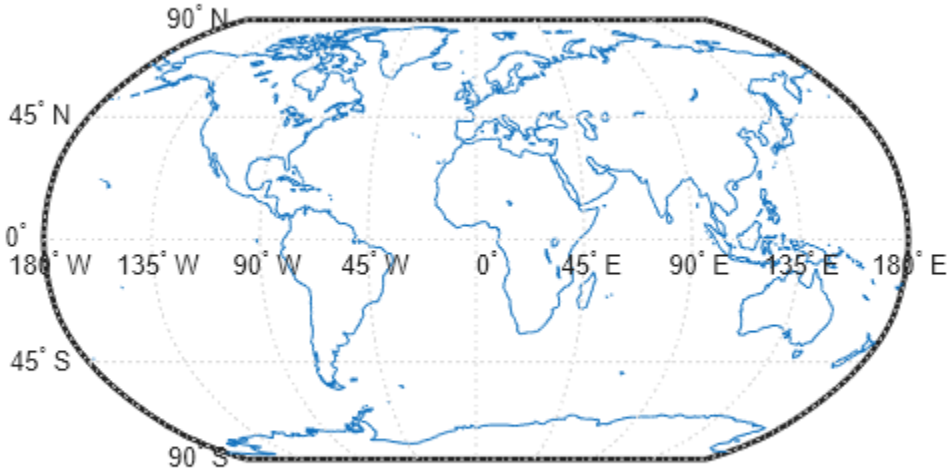
Set up a map of the world by using the `worldmap` function. The function automatically selects the map projection and the coordinate limits based on the region you specify. When you specify the region as `world`, the function selects a Robinson projection centered on the prime meridian and the equator.

```
figure
worldmap world
```



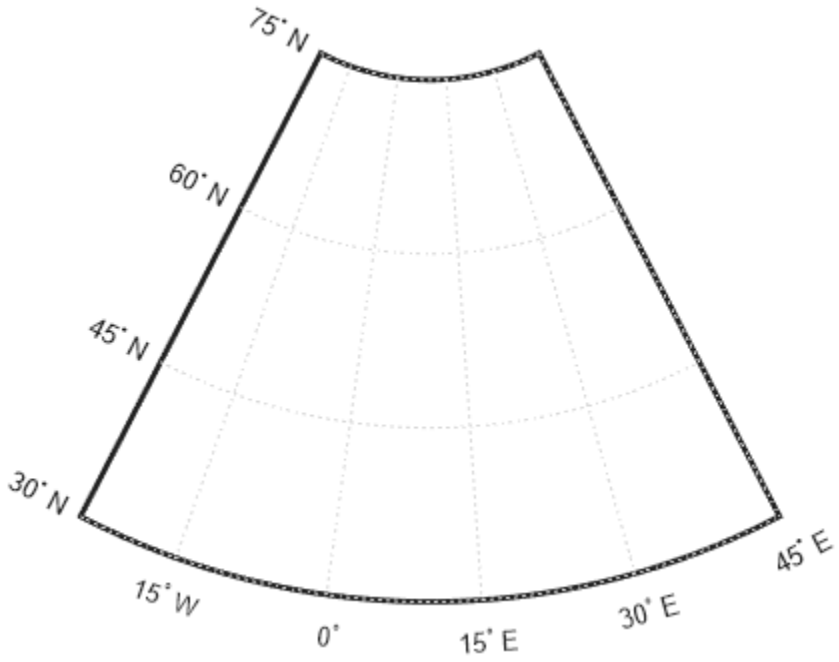
Display the coastline data using the `plotm` function.

```
load coastlines
plotm(coastlat,coastlon)
```



You can also create `axesm`-based maps for smaller regions. Set up a map of Europe by using the `worldmap` function.

```
figure
worldmap europe
```



In addition to standard axes properties, `axesm`-based maps contain properties such as the map projection type, the projection parameters, and the map limits. You can access and modify these additional properties using by using the `getm` and `setm` functions.

Query the map projection used by the map. The result indicates that the map uses an Equidistant Conic projection.

```
ax = gca;
getm(ax,"MapProjection")
```

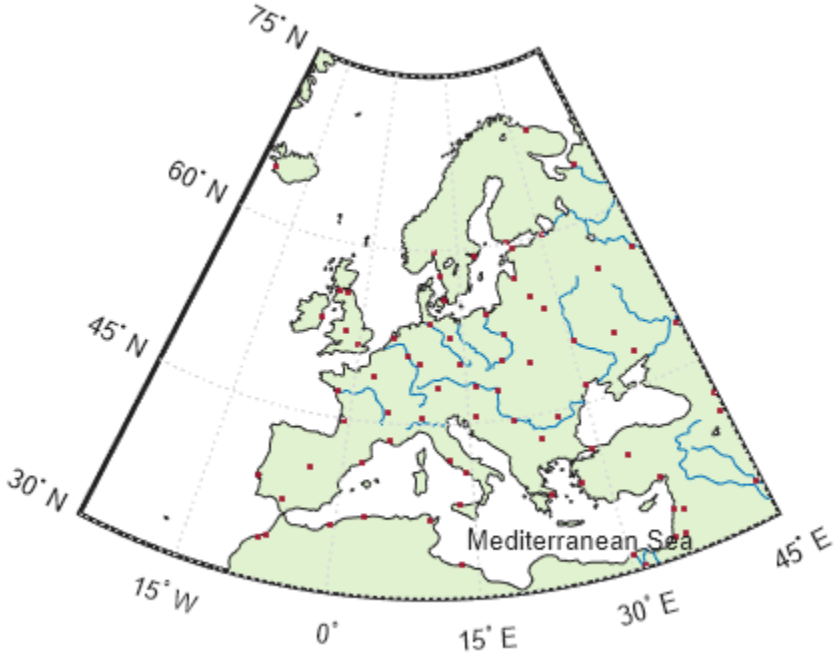
```
ans =
'eqdconic'
```

Display the data by using the `geoshow` function. Display the world land areas as green polygons, the world rivers as blue lines, and the world cities as red points.

```
geoshow(land,"FaceColor",[0.88 0.95 0.81])
geoshow(rivers,"Color",[0 0.4470 0.7410])
geoshow(cities,"Marker",".","MarkerEdgeColor",[0.6350 0.0780 0.1840])
```

Label the Mediterranean Sea by using the `textm` function. Specify the location of the text using latitude and longitude coordinates.

```
textm(35,14,"Mediterranean Sea")
```



See Also

Properties
MapAxes Properties | axesm-Based Map Properties

Related Examples

- “Create Common Plots Using Map Axes” on page 3-6
- “Create Map of Quadrangle Using Cartographic Map Layout” on page 4-16

What Is a Map?

Mapping Toolbox software manipulates electronic representations of geographic data. It lets you import, create, use, and present geographic data in various forms and to various ends. In the digital network era, it is easy to think of geospatial data as maps and maps as data, but you should take care to note the differences between these concepts.

The simplest (although perhaps not the most general) definition of a *map* is *a representation of geographic data*. Most people today generally think of maps as two-dimensional; to the ancient Egyptians, however, maps first took the form of lists of place names in the order they would be encountered when following a given road. Today such a list would be considered as *map data* rather than as a map. When most people hear the word "map" they tend to visualize two-dimensional renditions such as printed road, political, and topographic maps, but even classroom globes and computer graphic flight simulation scenes are maps under this definition.

In this toolbox, map data is any variable or set of variables representing a set of geographic locations, properties of a region, or features on a planet's surface, regardless of how large or complex the data is, or how it is formatted. Such data can be rendered as maps in a variety of ways using the functions and user interfaces provided.

What Is Geospatial Data?

Geospatial data comes in many forms and formats, and its structure is more complicated than tabular or even nongeographic geometric data. It is, in fact, a subset of spatial data, which is simply data that indicates where things are within a given *coordinate system*. Mileposts on a highway, an engineering drawing of an automobile part, and a rendering of a building elevation all have coordinate systems, and can be represented as spatial data when properly quantified (digitized). Such coordinate systems, however, are local and not explicitly tied or oriented to the Earth's surface; thus, most digital representations of mileposts, machine parts, and buildings do not qualify as geospatial data (also called *geodata*).

What sets geospatial data apart from other spatial data is that it is absolutely or relatively positioned on a planet, or *georeferenced*. That is, it has a *terrestrial coordinate system* that can be shared by other geospatial data. There are many ways to define a terrestrial coordinate system and also to transform it to any number of local coordinate systems, for example, to create a map projection. However, most are based on a framework that represents a planet as a sphere or spheroid that spins on a north-south axis, and which is girded by an *equator* (an imaginary plane midway between the poles and perpendicular to the rotational axis).

Geodata is coded for computer storage and applications in two principal ways: *vector* and *raster* representations. It has been said that "raster is faster but vector is corrector." There is truth to this, but the situation is more complex. For more information, see “Vector Geodata” on page 1-13 and “Raster Geodata” on page 1-14.

Vector Geodata

Vector data (in the computer graphics sense rather than the physics sense) can represent a map. Such vectors take the form of sequences of latitude-longitude or projected coordinate pairs representing a point set, a linear map feature, or an areal map feature. For example, points delineating the boundary of the United States, the interstate highway system, the centers of major U.S. cities, or even all three sets taken together, can be used to make a map. In such representations, the geographic data is in *vector* format and displays of it are referred to as *vector maps*. Such data consists of lists of specific coordinate locations (which, if describing linear or areal features, are normally points of inflection where line direction changes), along with some indication of whether each is connected to the points adjacent to it in the list.

In the Mapping Toolbox environment, vector data consists of sequentially ordered pairs of geographic (latitude, longitude) or projected (x,y) coordinate pairs (also called *tuples*). Successive pairs are assumed to be connected in sequence; breaks in connectivity must be delineated by the creation of separate vector variables or by inserting separators (usually NaNs) into the sets at each breakpoint. For vector map data, the connectivity (topological structure) of the data is often only a concern during display, but it also affects the computation of statistics such as length and area.

You can represent vector data by using geospatial tables or data structures. For more information about geospatial tables, see “Create Geospatial Tables” on page 6-2. For more information about data structures, see “Geographic Data Structures” on page 6-7.

Raster Geodata

You can map data represented as a *matrix* (a 2-D MATLAB® array) in which each row-and-column element corresponds to a rectangular patch of a specific geographic area, with implied topological connectivity to adjacent patches. This is commonly referred to as *raster data*. *Raster* is actually a hardware term meaning a systematic scan of an image that encodes it into a regular grid of pixel values arrayed in rows and columns.

When data in raster format represents the surface of a planet, it is called a *data grid*, and the data is stored as an array or matrix. The toolbox leverages the power of MATLAB matrix manipulation in handling this type of map data. This documentation uses the terms *raster data* and *data grid* interchangeably to talk about geodata stored in two-dimensional array form.

A raster can encode either an average value across a cell or a value sampled (posted) at the center of that cell. While geolocated data grids explicitly indicate which type of values are present (see “Geolocated Data Grids” on page 6-53), external metadata/user knowledge is required to be able to specify whether a regular data grid encodes averages or samples of values.

Digital Elevation Data

When raster geodata consists of surface elevations, the map can also be referred to as a *digital elevation model/matrix* (DEM), and its display is a *topographical map*. The DEM is one of the most common forms of *digital terrain model* (DTM), which can also be represented as contour lines, triangulated elevation points, quadtrees, octree, or otherwise.

For example, a raster of global terrain data is an example of a DEM. Suppose that the raster is represented by a 180-by-360 matrix. Each row represents one degree of latitude, and each column represents one degree of longitude. Each element of this matrix is the average elevation, in meters, for the one-degree-by-one-degree region of the Earth to which its row and column correspond.

Remotely Sensed Image Data

Raster geodata also encompasses georeferenced imagery. Like data grids, images are organized into rows and columns. There are subtle distinctions, however, which are important in certain contexts. One distinction is that an image may contain RGB or multispectral channels in a single array, so that it has a third (color or spectral) dimension. In this case a 3-D array is used rather than a 2-D (matrix) array. Another distinction is that while data grids are stored as class double in the toolbox, images may use a range of MATLAB storage classes, with the most common being `uint8`, `uint16`, `double`, and `logical`. Finally, for grayscale and RGB images of class double, the values of individual array elements are constrained to the interval `[0 1]`.

In terms of georeferencing—converting between column/row subscripts and 2-D map or geographic coordinates—images and data grids behave the same way (which is why both are considered to be a form of raster geodata). However, when performing operations that process the values raster elements themselves, including most display functions, it is important to be aware of whether you are working with an image or a data grid, and for images, how spectral data is encoded.

Latitude and Longitude

Two angles, latitude and longitude, specify the position of a point on the surface of a planet. The most common angular units are degrees and radians. Most Mapping Toolbox functions perform angle computations in degrees.

The latitude of a point is the angle between the plane of the equator and a line that connects the point to the rotational axis of the planet. The type and value of the latitude depends on the way you define the line. Latitude is positive in the northern hemisphere, reaching a limit of +90° at the north pole, and negative in the southern hemisphere, reaching a limit of –90° at the south pole. Lines of constant latitude are called parallels.

The longitude of a point is the angle at the center of the planet between two planes that align with and extend from the axis of rotation, perpendicular to the plane of the equator. One plane passes through the point, and the other plane is the prime meridian (0° longitude), which is defined by the location of the Royal Observatory in Greenwich, England. Lines of constant longitude are called *meridians*. All meridians converge at the north and south poles. As a result, longitude is underspecified at the poles.

Longitudes typically range from –180° to +180°, though you can also use ranges such as 0° to +360°. Adding or subtracting 360° to or from the longitude of a point does not change the position of the point. To convert longitudes from one range to another, use the `wrapTo180`, `wrapTo360`, `wrapToPi`, or `wrapTo2Pi` function.

Plot Latitude and Longitude

Load a MAT file containing the coordinates of global coastlines. The variables within the MAT file, `coastlat` and `coastlon`, specify numeric latitude and longitude coordinates, respectively.

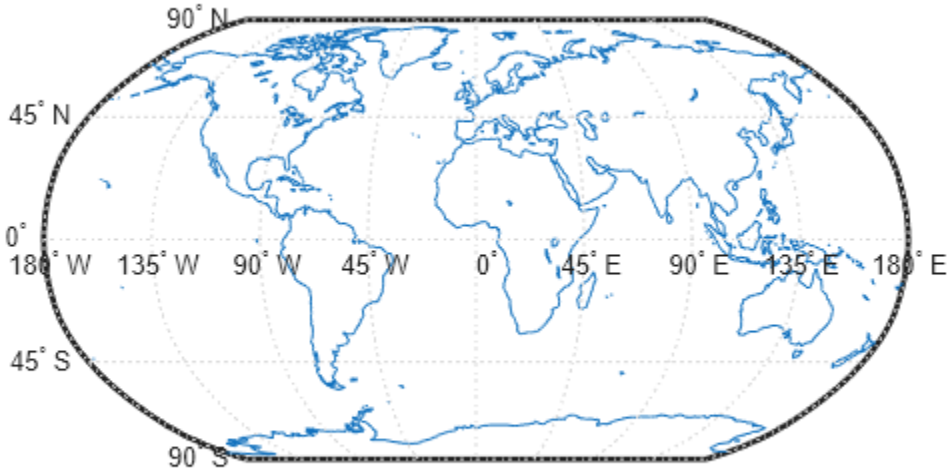
```
load coastlines
```

Set up a map of the world by using the `worldmap` function. The function automatically selects the map projection and the coordinate limits based on the region you specify.

```
figure
worldmap world
```

Display the coastline data by using the `plotm` function.

```
plotm(coastlat,coastlon)
```



See Also

Functions
`worldmap` | `plotm`

Related Examples

- “Choose a 2-D Map Display” on page 3-2
- “Create Your First World Map” on page 1-4
- “Angle Representations and Angular Units” on page 8-30

Data Import and Export

- “Supported Geospatial File Formats for Import and Export” on page 2-2
- “Mapping Toolbox Data Sets” on page 2-4
- “Find Geospatial Vector Data” on page 2-8
- “Find Geospatial Raster Data” on page 2-9
- “Functions That Read and Write Geospatial Data” on page 2-11
- “Data Stored in OpenStreetMap Layers” on page 2-13
- “Read Data from OpenStreetMap Files” on page 2-21
- “Display Buildings from OpenStreetMap Files” on page 2-27
- “Export Vector Geodata” on page 2-33
- “Export KML Files for Viewing in Earth Browsers” on page 2-34
- “Select Shapefile Data to Read” on page 2-39
- “Exporting Vector Data to KML” on page 2-43
- “Export Images and Raster Grids to GeoTIFF” on page 2-53
- “Create, Process, and Export Digital Surface Model from Lidar Data” on page 2-66
- “Visualize Sea Ice Concentrations from GRIB Data” on page 2-74
- “Convert Coastline Data (GSHHG) to Shapefile Format” on page 2-82

Supported Geospatial File Formats for Import and Export

This topic shows the geospatial file formats that you can import and export using Mapping Toolbox. For information about supported file formats that do not contain geospatial information, see "Supported File Formats for Import and Export".

Mapping Toolbox enables you to import and export data in raster and vector formats. Raster data stores information as arrays. Vector data stores information as points, lines, and polygons, as well as their nonspatial attributes.

Supported Import Formats

Raster Data

Read a geospatial raster data file by using the `readgeoraster` function. The `readgeoraster` function supports these file formats.

- GeoTIFF (`.tif` or `.tiff`)
- Esri Binary Grid (`.adf`)
- Esri ASCII Grid (`.asc` or `.grd`)
- Esri GridFloat (`.flt`)
- GRIB (`.grb`, `.grib`, `.grib2`)
- DTED (`.dt0`, `.dt1`, or `.dt2`)
- SDTS (`.DDF`)
- USGS DEM (`.dem`)
- SRTM Height (`.hgt`)
- Vertical Mapper Numeric Grid (`.grd`)
- Vertical Mapper Classified Grid (`.grc`)
- ER Mapper ERS (`.ers`)
- ENVI (`.dat`)
- ERDAS IMAGINE (`.img`)
- Geospatially referenced JPEG 2000 (`.jp2`)

The extension of a file does not always indicate its file format. If you do not know the format of your file, ask your data provider. In some cases, you can read files in supported formats when they have no extensions or have extensions other than the ones listed.

Read an image file that is spatially referenced by a world file by using the `imread` and `worldfileread` functions. Read the image file using `imread` and read the world file using `worldfileread`. Examples of world file extensions are `.tfw`, `.jpegw`, `.wld`, and `.pgw`.

Vector Data

Read a geospatial vector data file by using the `readgeotable` function. The `readgeotable` function supports these file formats.

- Esri file geodatabase (`.gdb`)

- GeoJSON (`.json` or `.geojson`)
- GeoPackage (`.gpkg`)
- GPX (`.gpx`)
- KML (`.kml`)
- KMZ (`.kmz`)
- OpenStreetMap® (`.osm` or `.osm.pbf`)
- Shapefile (`.shp`)

Supported Export Formats

Raster Data

Export raster data from the MATLAB workspace into a GeoTIFF file by using the `geotiffwrite` function.

Export raster data from the MATLAB workspace into an MBTiles file by using the `mbtileswrite` function.

Vector Data

Export vector data from the MATLAB workspace into a shapefile or KML file by using one of these methods:

- Export data into a shapefile by using the `shapewrite` function.
- Export data stored in numeric vectors into a KML file by using the `kmlwritepoint`, `kmlwriteline`, or `kmlwritepolygon` function.
- Export data stored in another representation, such as a geospatial table, into a KML file by using the `kmlwrite` function.

Tips

- Data vendors might refer to file formats using different names. For example, vendors might refer to the Esri Binary Grid format as ArcGrid Binary, Esri ArcGIS Binary Grid, or Esri ArcInfo Grid.
- Data vendors often provide data in multiple formats. If your data is not in a supported format, try getting the data in a different format.

See Also

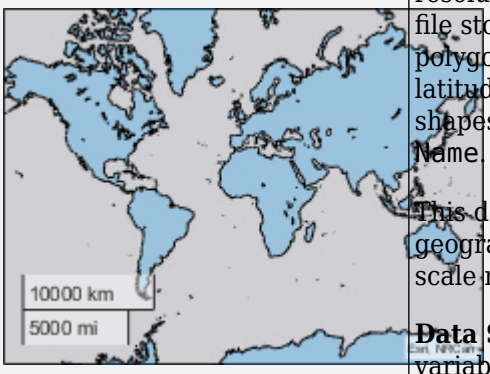
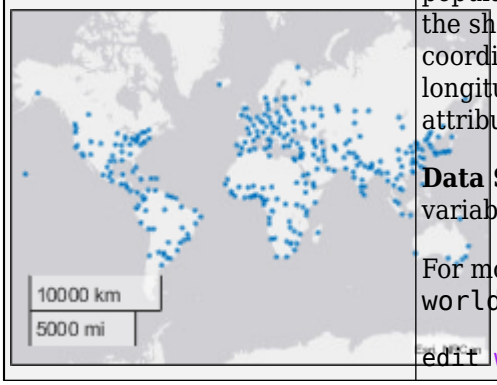
Related Examples

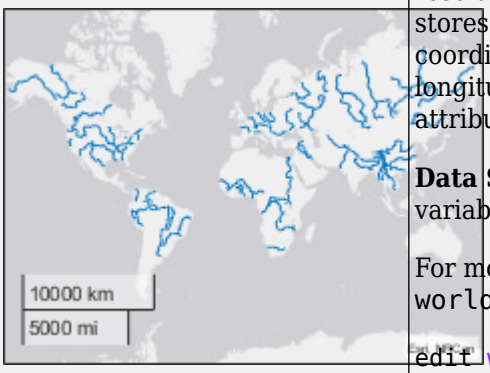
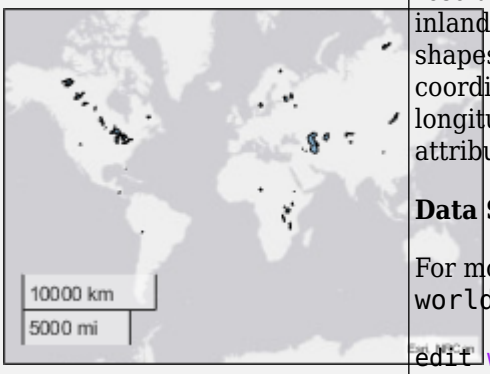
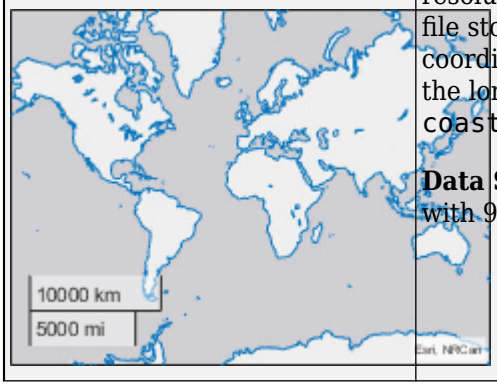
- “Find Geospatial Raster Data” on page 2-9
- “Find Geospatial Vector Data” on page 2-8
- “Functions That Read and Write Geospatial Data” on page 2-11

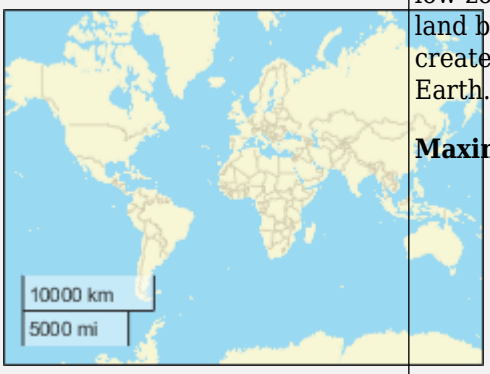
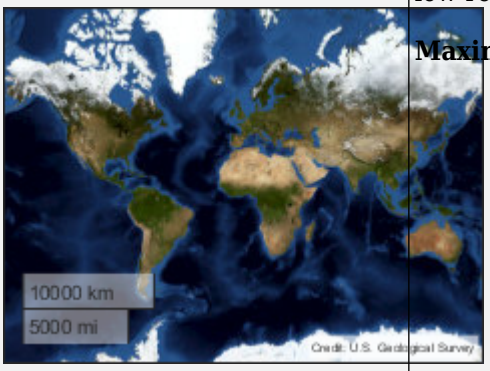
Mapping Toolbox Data Sets

Mapping Toolbox includes several files that contain data sets for the world and the United States. This topic summarizes useful data sets, but is not a comprehensive list.

World Data

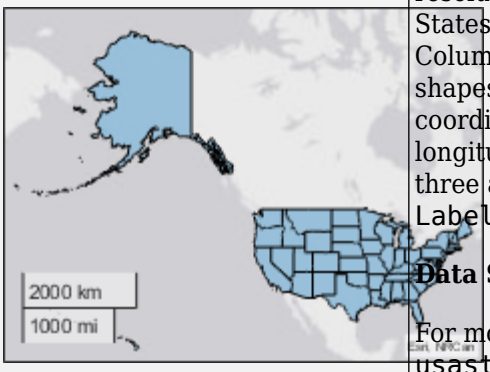
Filename	Description	How to Load
`landareas.shp` 	Shapefile containing low-resolution world land areas. The file stores the shapes as polygons with coordinates in latitude and longitude. The shapes have one attribute: `Name`. This data is useful for providing geographic context on small-scale maps. **Data Size:** 537 rows, 2 variables For more information, see `landareas.txt`. `edit landareas.txt`	`GT = readgeotable("landareas.shp");` `geoplot(GT)`
`worldcities.shp` 	Shapefile containing world populated places. The file stores the shapes as points with coordinates in latitude and longitude. The shapes have one attribute: `Name`. **Data Size:** 318 rows, 2 variables For more information, see `worldcities.txt`. `edit worldcities.txt`	`GT = readgeotable("worldcities.shp");` `geoplot(GT)`

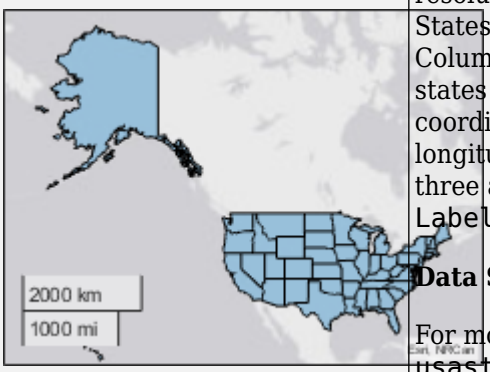
Filename	Description	How to Load
`worldrivers.shp` 	Shapefile containing low-resolution world rivers. The file stores the shapes as lines with coordinates in latitude and longitude. The shapes have one attribute: `Name`. **Data Size:** 128 rows, 2 variables For more information, see `worldrivers.txt`. `edit worldrivers.txt`	`GT = readgeotable("worldrivers.shp");` `geoplot(GT)`
`worldlakes.shp` 	Shapefile containing low-resolution world lakes and inland seas. The file stores the shapes as polygons with coordinates in latitude and longitude. The shapes have one attribute: `Name`. **Data Size:** 37 rows, 2 variables For more information, see `worldlakes.txt`. `edit worldlakes.txt`	`GT = readgeotable("worldlakes.shp");` `geoplot(GT)`
`coastlines.mat` 	MAT file containing low-resolution global coastlines. The file stores the latitude coordinates in `coastlat` and the longitude coordinates in `coastlon`. **Data Size:** Two column vectors with 9865 elements	`load coastlines.mat` `geoplot(coastlat,coastlon)`

Filename	Description	How to Load
`naturalearth.mbtiles` 	Vector MBTiles file containing low-zoom levels of region and land boundaries.[a] The file was created using data from Natural Earth. **Maximum Zoom Level:** 5	`addCustomBasemap("naturalearth","natura` `geobasemap naturalearth`
`usgsimagery.mbtiles` 	Raster MBTiles file containing low-resolution USGS imagery. **Maximum Zoom Level:** 4	`addCustomBasemap("usgsimagery","usgsima` `geobasemap usgsimagery`

a Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

United States Data

Filename	Description	How to Load
`usastatehi.shp` 	Shapefile containing moderate-resolution outlines of the United States and the District of Columbia. The file stores the shapes as polygons with coordinates in latitude and longitude. The shapes have three attributes: `Name`, `LabelLat`, and `LabelLon`. **Data Size:** 51 rows, 4 variables For more information, see `usastatehi.txt`. `edit usastatehi.txt`	`GT = readgeotable("usastatehi.shp");` `geoplot(GT)`

Filename	Description	How to Load
`usastatelo.shp` 	Shapefile containing low-resolution outlines of the United States and the District of Columbia. The file stores the states as polygons with coordinates in latitude and longitude. The shapes have three attributes: `Name`, `LabelLat`, and `LabelLon`. **Data Size:** 51 rows, 4 variables For more information, see `usastatelo.txt`. `edit usastatelo.txt`	`GT = readgeotable("usastatelo.shp");` `geoplot(GT)`

See Also

Functions

`readgeotable` | `addCustomBasemap` | `load`

Related Examples

- “MATLAB Example Data Sets”

Find Geospatial Vector Data

Find and download geospatial vector (point, line, or polygon) data using resources such as the ones in these tables. For information about supported file formats, see “Supported Geospatial File Formats for Import and Export” on page 2-2.

Resource	Provider	Examples of Products and Data Sets
OpenStreetMap	OpenStreetMap	Crowd-sourced map data for the world
The National Map Download Application	US Geological Survey (USGS)	USGS National Boundary Dataset
Advanced Weather Interactive Processing System (AWIPS) Basemap Shapefiles	National Oceanic and Atmospheric Administration (NOAA)	Canada provincial boundaries (data link), Mexico state boundaries (data link)
Geography Program	US Census	TIGER/Line shapefiles and geodatabases, cartographic boundary files
Data.gov	US General Services Administration	Agriculture, climate, energy, local government
NOAA Shoreline Website	NOAA	US coastlines, historical data
GSHHG — A Global Self-Consistent, Hierarchical, High-Resolution Geographic Database	University of Hawaii	Global Self-Consistent Hierarchical High-Resolution Geography (GSHHG) (read using the `gshhs` function)

See Also

Functions
`readgeotable` | `readtable` | `shaperead` | `shapeinfo`

See Also

More About

- “Find Geospatial Raster Data” on page 2-9
- “Supported Geospatial File Formats for Import and Export” on page 2-2

Find Geospatial Raster Data

Get geospatial raster data over the internet by downloading it or by accessing the Web Map Service (WMS) database.

Download Data

Find and download geospatial raster data using resources such as the ones in these tables. For information about supported file formats, see “Supported Geospatial File Formats for Import and Export” on page 2-2.

Elevation

Resource	Provider	Examples of Products and Data Sets
EarthExplorer	US Geological Survey (USGS)	DTED, GMTED2010, GTOPO30
The National Map Download Application	USGS	3DEP
Data.gov	US General Services Administration	DTED, 3DEP, GMTED2010
ETOPO Global Relief Model	National Oceanic and Atmospheric Administration (NOAA) and National Centers for Environmental Information (NCEI)	ETOPO 2022 (use GeoTIFF format)
GMTED2010 Viewer	USGS	GMTED2010
The Global Land One-km Base Elevation Project	NOAA and NCEI	GLOBE
Global Topography	Scripps Institution of Oceanography	Smith and Sandwell
OpenTopography	US National Science Foundation	High-Resolution Topography, 3DEP

Land Cover Classification

Resource	Provider	Examples of Products and Data Sets
EarthExplorer	USGS	GLCC, AVHRR
Digital Coast Data Access Viewer	NOAA Office for Coastal Management	C-CAP Regional Land Cover and Change
Data.gov	US General Services Administration	GLCC, AVHRR

Imagery and Spatial Context

Resource	Provider	Examples of Products and Data Sets
EarthExplorer	USGS	Landsat
Digital Coast Data Access Viewer	NOAA Office for Coastal Management	High-Resolution Orthoimagery
Data.gov	US General Services Administration	Landsat
Natural Earth	Natural Earth	Cross-Blended Hypsometric Tints, Shaded Relief, Bathymetry

Use Web Map Service Data

Mapping Toolbox includes a built-in database of prequalified Web Map Service (WMS) servers and layers. Search the WMS database for layers using the `wmsfind` function. Read layers from the database using the `wmsread` function.

See Also

`readgeoraster` | `worldfileread`

More About

- “Find Geospatial Vector Data” on page 2-8
- “Supported Geospatial File Formats for Import and Export” on page 2-2

Functions That Read and Write Geospatial Data

The following table lists Mapping Toolbox functions that read geospatial data products and file formats and write geospatial data files. Note that the `geoshow` and `mapshow` functions can read and display both vector and raster geodata files in several formats. Click function names to see their details in the Mapping Toolbox reference documentation. The **Type of Coordinates** column describes whether the function returns or writes data in geographic ("geo") or projected ("map") coordinates, or as geolocated data grids (which, for the functions listed, all contain geographic coordinates). Some functions can return either geographic or map coordinates, depending on what the file being read contains; these functions do not signify what type of coordinates they return (in the case of `shaperead`, however, you can specify whether the structure it returns should have `X` and `Y` or `Lon` and `Lat` fields).

Function	Description	Type of Data	Type of Coordinates
`avhrrgoode`	Read data products derived from the Advanced Very High Resolution Radiometer (AVHRR) and stored in the Goode Homosoline projection: Global Land Cover Classification (GLCC) or Normalized Difference Vegetation Index (NDVI)	raster	geolocated
`avhrrlambert`	Read AVHRR GLCC and NDVI data products stored in the Lambert Conformal Conic projection	raster	geolocated
`dteds`	List DTED data grid file names for a specified latitude-longitude quadrangle	file names	geo
`egm96geoid`	Read 15-minute gridded geoid heights from the EGM96 geoid model	raster	geo
`georasterinfo`	Get information about data files in formats such as Esri Binary Grid, Esri GridFloat, DTED, GeoTIFF, and GPX	raster	map geo
`geotiffinfo`	Get information about GeoTIFF files	properties	map geo
`geotiffwrite`	Write GeoTIFF file	raster	map geo
`getworldfilename`	Derive a world file name from an image file name	file name	geo map
`globedems`	List GLOBE data file names for a specified latitude-longitude quadrangle	file names	geo
`gshhs`	Read Global Self-Consistent Hierarchical High-Resolution Geography (GSHHG) data	vector	geo
`gtopo30s`	List GTOPO30 data file names for a specified latitude-longitude quadrangle	file names	geo

Function	Description	Type of Data	Type of Coordinates
`kmlwrite`	Write vector coordinates and attributes to a file in KML format	vector points and attributes	geo
`readgeoraster`	Read data in formats such as Esri Binary Grid, Esri GridFloat, DTED, GeoTIFF, and GPX	raster	geo map
`readgeotable`	Read data in formats such as shapefile, GPX, Esri file geodatabase, KML, and GeoJSON	vector	geo map
`shapeinfo`	Get information about the geometry and attributes of geographic features stored in a shapefile (a set of ".shp", ".shx" and ".dbf" files)	properties	geo map
`shaperead`	Read geographic feature coordinates and associated attributes from a shapefile	vector	geo map
`shapewrite`	Write geospatial data and associated attributes in shapefile format	vector	geo map
`usgsdems`	List USGS digital elevation model (DEM) file names covering a specified latitude-longitude quadrangle	file names	map
`vmap0data`	Extract selected data from the Vector Map Level 0 (VMAP0) CD-ROMs	vector	geo
`vmap0read`	Read a VMAP0 file	vector	geo
`vmap0rhead`	Read VMAP0 file headers	properties	geo
`worldfileread`	Read a world file and return a referencing matrix	georeferencing information	geo
`worldfilewrite`	Export a referencing matrix into an equivalent world file	georeferencing information	geo

The MATLAB environment provides many general file reading and writing functions (for example, `imread`, `imwrite`, `urlread`, and `urlwrite`) which you can use to access geospatial data you want to use with Mapping Toolbox software. For example, you can read a TIFF image with `imread` and its accompanying world file with `worldfileread` to import the image and construct a referencing matrix to georeference it. See the Mapping Toolbox example “Georeference an Image to an Orthotile Base Layer” on page 6-63 for an example of how you can do this.

See Also

Related Examples

- “Supported Geospatial File Formats for Import and Export” on page 2-2

Data Stored in OpenStreetMap Layers

OpenStreetMap files provide access to crowd-sourced map data from all over the world. The files store information about locations using tags. Each tag consists of a key and a value.

- The key provides context for the location. OpenStreetMap supports thousands of keys, such as `highway`, `building`, and `address`.
- The value provides detail about the key. Each key can support hundreds of values. For example, values for the `highway` key include `"yes"`, `"primary"`, `"footway"`, and `"cycleway"`.

When you read data from an OpenStreetMap file by using the `readgeotable` function, the function stores the data using a geospatial table. A geospatial table is a `table` or `timetable` object with a `Shape` variable and attribute variables.

The data that the `readgeotable` function reads depends on the value of the `Layer` name-value argument. This topic describes the data stored in each layer of an OpenStreetMap file.

Data Layers

Read data from an OpenStreetMap file by specifying the `Layer` name-value argument of the `readgeotable` function as `"points"`, `"lines"`, `"multilinestrings"`, or `"multipolygons"`. When you read data from an OpenStreetMap file, the geospatial table stores the tags using the attribute variables. For common tags, the names of the attribute variables match the keys.

Most locations stored in OpenStreetMap files do not have values for all the common tags. When a location does not contain a value for a tag, the geospatial table stores the value as `""`.

"points"

When you specify the `Layer` name-value argument as `"points"`, the `readgeotable` function reads standalone node features from the file. The output geospatial table has these variables.

Variable	Description
`Shape`	Point shape, returned as a `geopointshape` object.
`osm_id`	OpenStreetMap ID, returned as a string scalar.
`name`	Primary name, returned as a string scalar.
`barrier`	Barrier, returned as a string scalar. The points layer includes barriers such as gates, toll booths, and border control points.
`highway`	Highway, returned as a string scalar. The points layer includes highway elements such as crosswalks, stop signs, and traffic signals.
`ref`	Reference number or code, returned as a string scalar.
`address`	Address, returned as a string scalar.

Variable	Description
`is_in`	Descriptive location, returned as a string scalar. For example, this attribute can describe the city in which a point is located.
`place`	Type of place, returned as a string scalar. The points layer can label places as countries, states, or counties.
`man_made`	Man-made structure, returned as a string scalar. The points layer includes man-made structures such as windmills, antennas, and survey points.
`other_tags`	Other tags, returned as a string scalar. This variable stores the keys and values of OpenStreetMap tags that are not commonly used for points.
`other_tags_dictionary`	Other tags, returned as a cell array containing a `dictionary`. This variable stores the same content as `other_tags` in a more convenient form. For information on how to extract values stored in `other_tags_dictionary`, see the “Read Data from OpenStreetMap Files” on page 2-21 example.

"lines"

When you specify the `Layer` name-value argument as `"lines"`, the `readgeotable` function reads way features with no area from the file. The output geospatial table has these variables.

Variable	Description
`Shape`	Line shape, returned as a `geolineshape` object.
`osm_id`	OpenStreetMap ID, returned as a string scalar.
`name`	Primary name, returned as a string scalar.
`highway`	Highway, returned as a string scalar. The lines layer includes highway elements such as roads, footpaths, and cycleways.
`waterway`	Waterway, returned as a string scalar. The lines layer includes waterways such as rivers, drains, and dams.
`aerialway`	Aerialway, returned as a string scalar. The lines layer includes aerialways such as gondolas, chair lifts, and zip lines.
`barrier`	Barrier, returned as a string scalar. The lines layer includes barriers such as walls, fences, and guardrails.
`man_made`	Man-made structure, returned as a string scalar. The lines layer includes man-made structures such as pipelines, breakwaters, and embankments.

Variable	Description
`railway`	Railway, returned as a string scalar. The lines layer includes railway elements such as subways and light rail routes.
`other_tags`	Other tags, returned as a string scalar. This variable stores the keys and values of OpenStreetMap tags that are not commonly used for lines.
`other_tags_dictionary`	Other tags, returned as a cell array containing a `dictionary`. This variable stores the same content as `other_tags` in a more convenient form. For information on how to extract values stored in `other_tags_dictionary`, see the “Read Data from OpenStreetMap Files” on page 2-21 example.

"multilinestrings"

When you specify the `Layer` name-value argument as `"multilinestrings"`, the `readgeotable` function reads relation features with a multilinestring or route type from the file. The output geospatial table has these variables.

Variable	Description
`Shape`	Line shape, returned as a `geolineshape` object.
`osm_id`	OpenStreetMap ID, returned as a string scalar.
`name`	Primary name, returned as a string scalar.
`type`	Type, returns as a string scalar.
`other_tags`	Other tags, returned as a string scalar. This variable stores the keys and values of OpenStreetMap tags that are not commonly used for multilinestrings.
`other_tags_dictionary`	Other tags, returned as a cell array containing a `dictionary`. This variable stores the same content as `other_tags` in a more convenient form. For information on how to extract values stored in `other_tags_dictionary`, see the “Read Data from OpenStreetMap Files” on page 2-21 example.

"multipolygons"

When you specify the `Layer` name-value argument as `"multipolygons"`, the `readgeotable` function reads way features with area, and relationship features with a multipolygon or boundary type, from the file. The output geospatial table has these variables.

Variable	Description
`Shape`	Polygon shape, returned as a `geopolyshape` object.

Variable	Description
`osm_id`	OpenStreetMap ID, returned as a string scalar.
`osm_way_id`	OpenStreetMap way ID, returned as a string scalar.
`name`	Primary name, returned as a string scalar.
`type`	Type, returns as a string scalar.
`aeroway`	Aeroway, returned as a string scalar. The multipolygons layer includes aeroway elements such as runways, airport terminals, and control towers.
`amenity`	Amenity, returned as a string scalar. The multipolygons layer includes amenities such as libraries, restaurants, and hospitals.
`admin_level`	Administrative level within a government hierarchy, returned as a string scalar.
`barrier`	Barrier, returned as a string scalar. The multipolygons layer includes barriers such as walls and toll booths.
`boundary`	Boundary, returned as a string scalar. The multipolygons layer includes boundaries such as administrative boundaries, political boundaries, and national park boundaries.
`building`	Building, returned as a string scalar. The multipolygons layer includes buildings such as houses, commercial buildings, and retail shops.
`craft`	Workplace for creating craft goods, returned as a string scalar. The multipolygons layer includes workplaces for occupations such as electrician, shoemaker, and baker.
`geological`	Geological feature, returned as a string scalar. The multipolygons layer includes geological features such as lava fields and craters.
`historic`	Historic site, returned as a string scalar. The multipolygons layer includes historic sites such as buildings, farms, and battlefields.
`land_area`	Administrative boundary, returned as a string scalar.
`landuse`	Primary use of the land, returned as a string scalar. The multipolygons layer includes primary land uses such as education, industry, residence, and retail.
`leisure`	Location for leisure activity, returned as a string scalar. The multipolygons layer includes locations such as dog parks, ice rinks, and sports arenas.

Variable	Description
`man_made`	Man-made structure, returned as a string scalar. The multipolygons layer includes man-made structures such as bridges, courtyards, and satellite dishes.
`military`	Building or land used by the military, returned as a string scalar. The multipolygons layer includes military features such as bunkers and academies.
`natural`	Natural land feature, returned as a string scalar. The multipolygons layer includes natural land features such as grasslands, bays, and sinkholes.
`office`	Business office, returned as a string scalar. The multipolygons layer includes offices such as financial offices, government offices, and telecommunication offices.
`place`	Type of place, returned as a string scalar. The multipolygons layer includes place values such as state, region, and province.
`shop`	Shop, returned as a string scalar. The multipolygons layer includes shops such as bakeries, shopping malls, and car repair shops.
`sport`	Location for playing a sport, returned as a string scalar. The multipolygons layer includes sport locations such as baseball fields, rock climbing gyms, and weightlifting gyms.
`tourism`	Location of interest to tourists, returned as a string scalar. The multipolygons layer includes tourist locations such as campsites, theme parks, and art galleries.
`other_tags`	Other tags, returned as a string scalar. This variable stores the keys and values of OpenStreetMap tags that are not commonly used for points.
`other_tags_dictionary`	Other tags, returned as a cell array containing a `dictionary`. This variable stores the same content as `other_tags` in a more convenient form. For information on how to extract the values stored in `other_tags_dictionary`, see the “Read Data from OpenStreetMap Files” on page 2-21 example.

Building Layers

Read buildings from an OpenStreetMap file by specifying the `Layer` name-value argument of the `readgeotable` function as `"buildings"` or `"buildingparts"`. When you read buildings from an OpenStreetMap file, the geospatial table stores data that MATLAB derives from the tags.

"buildings"

When you specify the `Layer` name-value argument as `"buildings"`, the output geospatial table has these variables.

Variable	Description
`Shape`	Building footprint, returned as a `geopolyshape` object.
`Centroid`	Centroid of the building footprint, returned as a `geopointshape` object.
`Type`	Type, returned as a `map.Buildings.Type` enumeration with one of these values: • `standard` — Standard building that consists of one part with a flat roof. • `composite` — Composite building that consists of multiple parts. The table stores the parts in the `Parts` variable. • `roof` — Roof. The table stores the roof shape in the `RoofShape` variable and the roof direction in the `RoofDirection` variable.
`Parts`	Parts, returned as a one-element cell array containing a geospatial table. The geospatial table has the same variable names as the `"buildings"` layer.
`ID`	OpenStreetMap ID, returned as a string scalar.
`MinHeight`	Minimum height, returned as a scalar of type `double` in units of meters above ground.
`MaxHeight`	Maximum height, returned as a scalar of type `double` in units of meters above ground.
`Name`	Name, returned as a string scalar.
`Material`	Material, returned as a string scalar.
`Color`	Color, returned as a string scalar that represents a hexadecimal color code. A hexadecimal color code starts with a hash symbol (#) followed by three or six hexadecimal digits.
`RoofShape`	Roof shape, returned as a string scalar. This variable has a value only when the `Type` variable has a value of `"roof"`. Supported roof shapes include `"gabled"`, `"flat"`, `"hipped"`, and `"pyramidal"`.

Variable	Description
`RoofDirection`	Roof direction, returned as a string scalar. This variable has a value only when the `Type` variable has a value of `"roof"`. Supported roof directions include: • A compass direction, for example `"NW"` or `"northwest"`. • A number, in degrees, in the range [0, 360], for example `"180"`. • The direction of the roof ridge with respect to the longest building edge, for example `"along"` or `"across"`.

"buildingparts"

The `"buildingparts"` layer is a flattened version of the `"buildings"` layer. While the `"buildings"` layer stores building parts using the `Parts` variable, the `"buildingparts"` layer stores building parts using separate table rows.

When you specify the `Layer` name-value argument of the `readgeotable` function as `"buildingparts"`, the output geospatial table has these variables.

Variable	Description
`Shape`	Footprint of the building part, returned as a `geopolyshape` object.
`Centroid`	Centroid of the building part footprint, returned as a `geopointshape` object.
`Type`	Type, returned as a `map.Buildings.Type` enumeration with one of these values: • `standard` — Standard building that consists of one part with a flat roof. • `composite` — Composite building that consists of multiple parts. • `roof` — Roof. The table stores the roof shape in the `RoofShape` variable and the roof direction in the `RoofDirection` variable.
`BuildingID`	OpenStreetMap ID of the building, returned as a string scalar.
`PartID`	OpenStreetMap ID of the building part, returned as a string scalar.
`MinHeight`	Minimum height, returned as a scalar of type `double` in units of meters above ground.
`MaxHeight`	Maximum height, returned as a scalar of type `double` in units of meters above ground.

Variable	Description
`Name`	Name, returned as a string scalar.
`Material`	Material, returned as a string scalar.
`Color`	Color, returned as a string scalar that represents a hexadecimal color code. A hexadecimal color code starts with a hash symbol (#) followed by three or six hexadecimal digits.
`RoofShape`	Roof shape, returned as a string scalar. This variable has a value only when the `Type` variable has a value of `"roof"`. Supported roof shapes include `"gabled"`, `"flat"`, `"hipped"`, and `"pyramidal"`.
`RoofDirection`	Roof direction, returned as a string scalar. This variable has a value only when the `Type` variable has a value of `"roof"`. Supported roof directions include: • A compass direction, for example `"NW"` or `"northwest"`. • A number, in degrees, in the range [0,360], for example `"180"`. • The direction of the roof ridge with respect to the longest building edge, for example `"along"` or `"across"`.

See Also

`readgeotable`

Related Examples

- “Read Data from OpenStreetMap Files” on page 2-21
- “Display Buildings from OpenStreetMap Files” on page 2-27

External Websites

- OpenStreetMap

Read Data from OpenStreetMap Files

This example shows how to read OpenStreetMap® files into geospatial tables and then find locations stored in the table based on their tags.

OpenStreetMap files provide information about locations using *tags*. Each tag consists of a key and a value.

- The *key* provides context for the location. OpenStreetMap supports thousands of keys, such as `highway`, `building`, and `address`.
- The *value* provides detail about the key. Each key can support hundreds of values. For example, values for the `highway` key include `"yes"`, `"primary"`, `"footway"`, and `"cycleway"`.

A geospatial table is a `table` or `timetable` object that contains a `Shape` table variable and attribute table variables. When you read data from an OpenStreetMap file into a geospatial table by using the `readgeotable` function, the geospatial table stores the tags using the attribute table variables. For common tags, the names of the variables match the keys. For uncommon tags, the table stores the keys and values in the `other_tags_dictionary` variable. Most locations stored in OpenStreetMap files do not have values for all tags.

Read Data from File

Specify the name of an OpenStreetMap file [1] on page 2-25 containing data for several city blocks in Shibuya, Tokyo, Japan.

```
filename = "shibuya.osm";
```

Read the lines and points layers from the file into geospatial tables.

- The lines layer represents features such as roads, sidewalks, and railroad tracks. The table represents the lines using line shapes in geographic coordinates.
- The points layer represents features such as traffic signals, bus stops, and subway entrances. The table represents the points using point shapes in geographic coordinates.

```
linesLayer = readgeotable(filename,Layer="lines");
pointsLayer = readgeotable(filename,Layer="points");
```

Find Locations with Tag

Display lines and points with a `railway` tag from an OpenStreetMap file. OpenStreetMap files use the `railway` tag to specify information about railways. For example, a `railway` tag can identify a line as a subway route and a point as a station entrance.

Query Tags Stored in Table Variables

The lines layer includes the `railway` key as a table variable. You can verify that the table has a `railway` variable by using the `matches` function.

```
matches("railway",linesLayer.Properties.VariableNames)
```

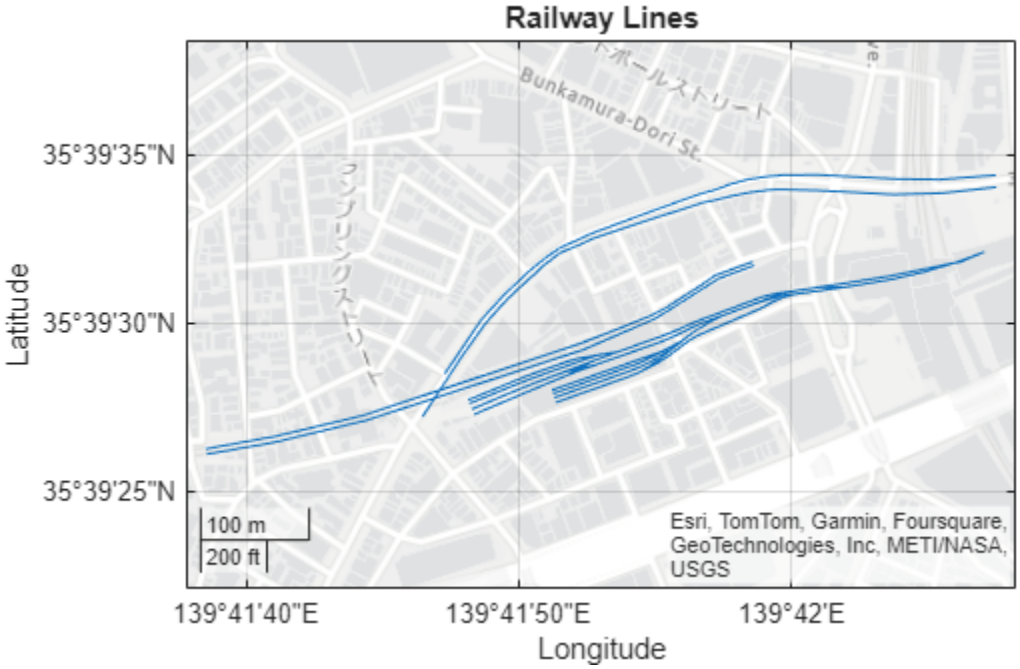
```
ans = logical
   1
```

Find the table rows that have `railway` tags. If a table row does not have a `railway` tag, then the value of the `railway` key for that row is `""`. Create a new geospatial table from the table rows with `railway` tags.

```
idxRailways = ~ismissing(linesLayer.railway,"");
railways = linesLayer(idxRailways,:);
```

Display the line shapes with `railway` tags on a map.

```
figure
geoplot(railways)
title("Railway Lines")
```



Query Tags Stored in `other_tags_dictionary`

The `railway` tag is less common for the points layer, so the points layer stores instances of the `railway` tag in the `other_tags_dictionary` variable. You can verify that the table does not have a `railway` variable by using the `matches` function.

```
matches("railway",pointsLayer.Properties.VariableNames)
```

```
ans = logical
   0
```

The `other_tags_dictionary` variable stores tags in cell arrays of dictionaries. Find the table rows that contain `railway` tags by using a loop. For each row of the table, get the dictionary from the cell array stored in the `other_tags_dictionary` variable. Then, determine whether the dictionary contains a `railway` tag.

```
numRows = height(pointsLayer);
hasRailwayTag = false(numRows,1);
for row = 1:numRows
```

```
    dict = pointsLayer.other_tags_dictionary{row};
    hasRailwayTag(row) = isKey(dict,"railway");
end
```

Verify that at least one of the table rows contains a `railway` tag.

```
any(hasRailwayTag)
```

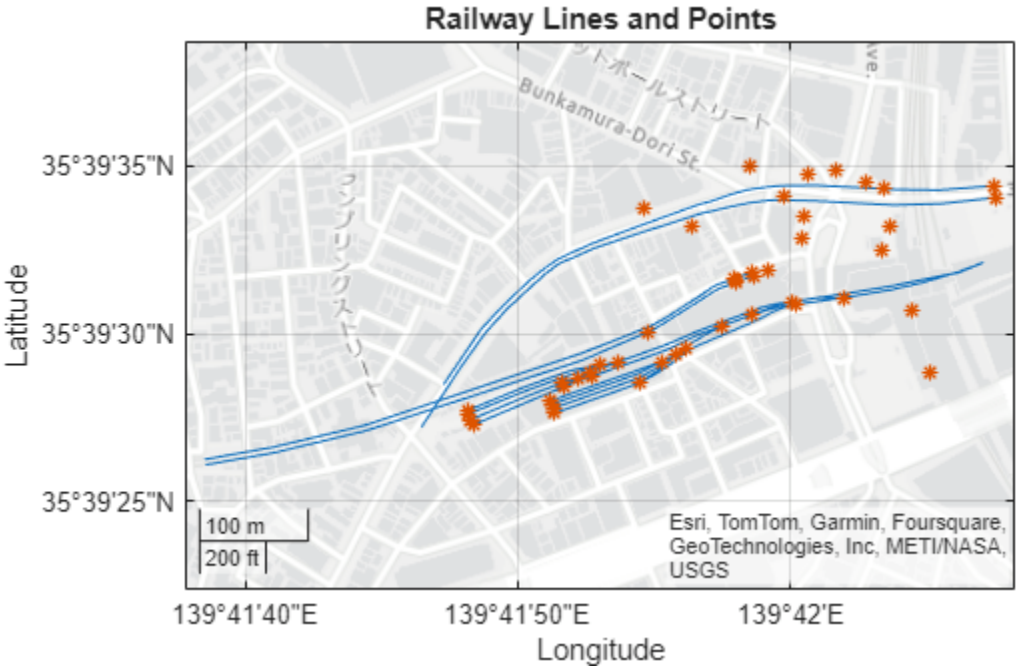
```
ans = logical
   1
```

Create a new geospatial table from the table rows with `railway` tags.

```
railpoints = pointsLayer(hasRailwayTag,:);
```

Display the points with `railway` tags on the same map as the lines.

```
hold on
geoplot(railpoints,"*")
title("Railway Lines and Points")
```



Find Tags with Specified Values

Display lines that represent footpaths and points that represent subway entrances. OpenStreetMap files identify footpaths using the `highway` key and the `"footway"` value, and they identify subway entrances using the `railway` key and the `"subway_entrance"` value.

Query Tags and Values Stored in Table Variables

The lines layer includes the `highway` key as a table variable. You can verify that the table has a `highway` variable by using the `matches` function.

```
matches("highway",linesLayer.Properties.VariableNames)
```

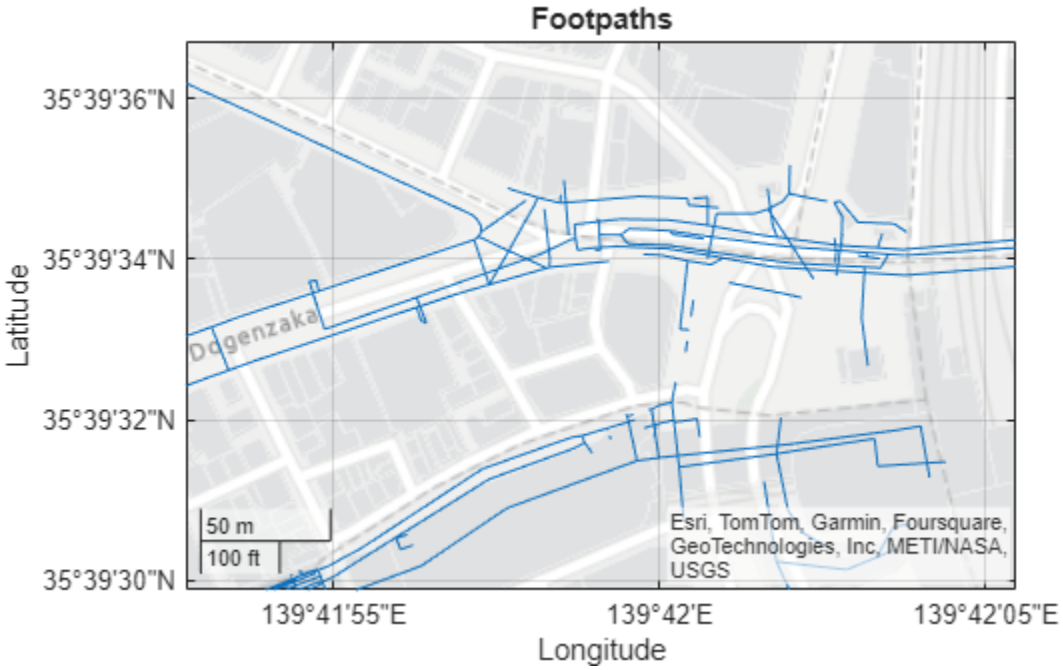
```
ans = logical
   1
```

Find the table rows that represent footpaths. Create a new geospatial table from the footpath table rows.

```
isFootpath = linesLayer.highway == "footway";
footway = linesLayer(isFootpath,:);
```

Display the footpaths on a map. Zoom in to the footpaths surrounding Shibuya Station.

```
figure
geoplot(footway)
geolimits([35.6583 35.6602],[139.698 139.7015])
title("Footpaths")
```



Query Tags and Values Stored in `other_tags_dictionary`

Extract instances of the `railway` tag from the `other_tags_dictionary` variable and store the values using a new table variable.

The points layer stores instances of the `railway` tag in the `other_tags_dictionary` variable. You can verify that the table does not have a `railway` variable by using the `matches` function.

```
matches("railway",pointsLayer.Properties.VariableNames)
```

```
ans = logical
   0
```

The `other_tags_dictionary` variable stores tags in cell arrays of dictionaries. Create a new table variable called `railway` that contains `""` values. Then, add data to the variable by using a loop. For

each row of the table, get the dictionary from the cell array stored in the `other_tags_dictionary` variable. If the dictionary contains the `railway` tag, add the value to the `railway` variable.

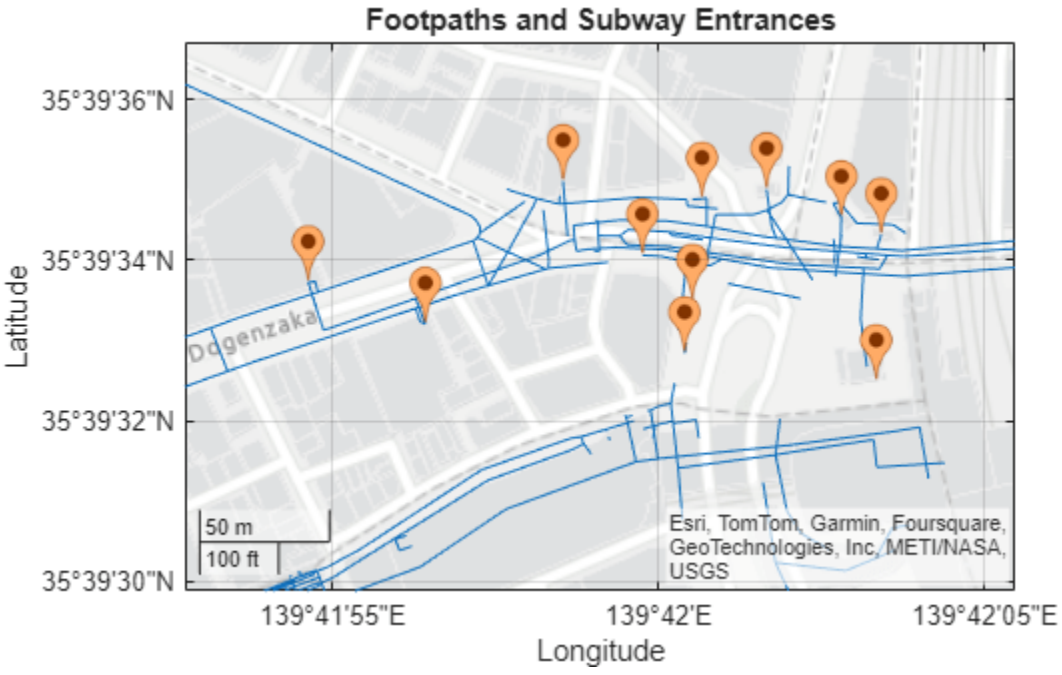
```
numRows = height(pointsLayer);
pointsLayer.railway = strings(numRows,1);
for row = 1:numRows
    dict = pointsLayer.other_tags_dictionary{row};
    hasTag = isKey(dict,"railway");
    if hasTag
        pointsLayer.railway(row) = dict("railway");
    end
end
```

Find the table rows that represent subway entrances by querying the `railway` variable. Create a new geospatial table from the table rows.

```
idxEntrance = pointsLayer.railway == "subway_entrance";
entrance = pointsLayer(idxEntrance,:);
```

Extract the latitude and longitude coordinates from the table. Then, on the same map, display the subway entrances using pushpin icons.

```
lat = entrance.Shape.Latitude;
lon = entrance.Shape.Longitude;

hold on
geoiconchart(lat,lon,SizeData=30)
title("Footpaths and Subway Entrances")
```



[1] You can download OpenStreetMap files from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

See Also

Functions
`readgeotable` | `matches` | `ismissing` | `isKey`

Objects
`dictionary`

Related Examples

- “Create Geospatial Tables” on page 6-2
- “Map Data with Dictionaries”
- “Display Buildings from OpenStreetMap Files” on page 2-27

Display Buildings from OpenStreetMap Files

This topic shows how to read building footprints from OpenStreetMap® files into geospatial tables, and then visualize the attributes of the buildings using colors and data tips.

When you read buildings from an OpenStreetMap file by using the `readgeotable` function, the function derives information about the buildings from the file and stores the result in a geospatial table. The geospatial table contains the building footprints, the footprint centroids, and information such as the building heights, colors, and materials.

Read Buildings from OpenStreetMap File

Read the buildings layer from an OpenStreetMap file [1] on page 2-31 containing data for several city blocks in Shibuya, Tokyo, Japan. The buildings within the file have edited colors and materials. The output geospatial table represents the buildings using polygon shapes in geographic coordinates.

```
buildingsLayer = readgeotable("shibuya.osm",Layer="buildings");
```

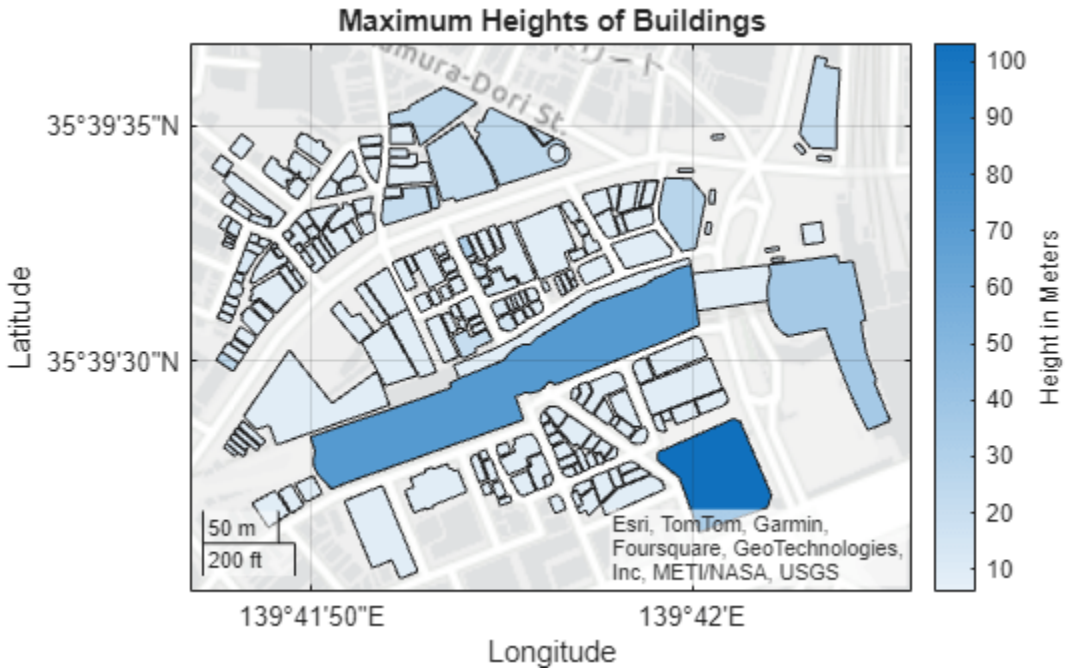
Visualize Maximum Heights of Buildings

The geospatial table stores the heights of the buildings in the `MaxHeight` table variable. Display the building footprints on a map by using the `geoplot` function. Vary the plot colors with the heights by specifying the `ColorVariable` name-value argument.

```
figure
geoplot(buildingsLayer,ColorVariable="MaxHeight")
```

Add a title, change the colormap, and add a labeled color bar.

```
title("Maximum Heights of Buildings")
colormap sky
c = colorbar;
c.Label.String = "Height in Meters";
```



Visualize Building Materials

The geospatial table stores the building materials in the `Material` table variable. Visualize the building footprints using a different color for each material.

When an OpenStreetMap file does not specify the material of a building, the table stores the `Material` value as `""`. For readability, find the table rows that do not specify a building material and replace `""` with `"unknown"`.

```
idxNoMaterial = buildingsLayer.Material == "";
buildingsLayer.Material(idxNoMaterial) = "unknown";
```

Find the unique building materials stored in the table.

```
materials = unique(buildingsLayer.Material)
```

```
materials = 4×1 string
    "concrete"
    "glass"
    "stone"
    "unknown"
```

Display the building footprints on a road map. To create a legend from the materials, display the building footprints for each material using a loop. Within the loop:

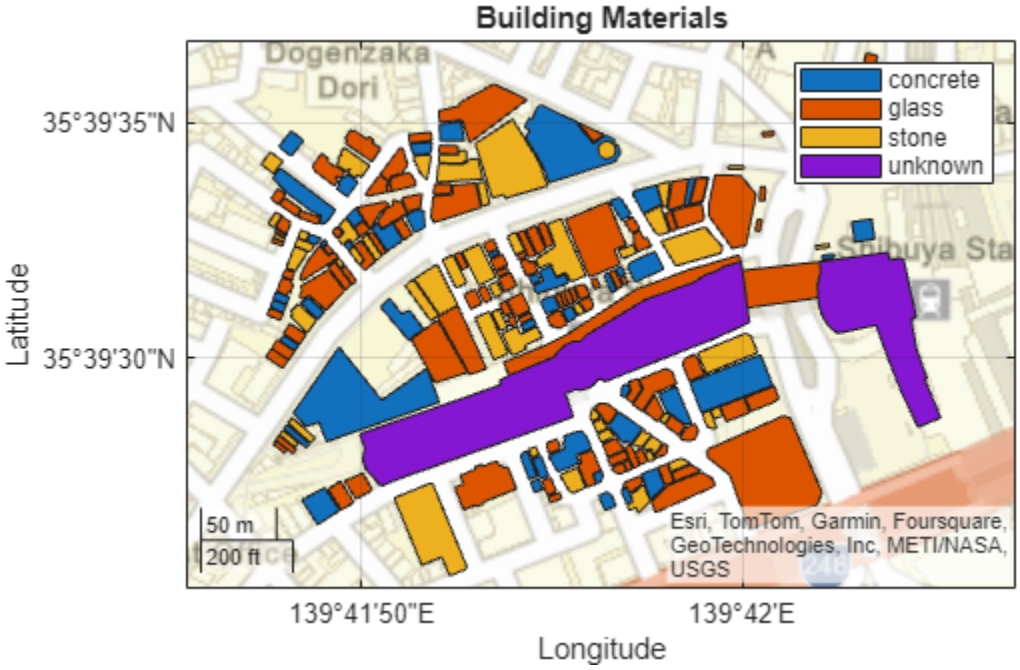
- Get a material from the list of unique materials.
- Find the table rows that use the material.
- Create a subtable from the table rows.
- Display the buildings from the subtable.

```
figure
geobasemap streets
```

```
hold on

for k = 1:length(materials)
   m = materials(k);
   idxm = buildingsLayer.Material == m;
   rows = buildingsLayer(idxm,:);
   geoplot(rows,DisplayName=m);
end
```

Add a title and legend, and make the buildings more opaque.

```
title("Building Materials")
legend
alpha(1)
```



Visualize Building Colors

The geospatial table stores the building colors in the `Color` table variable. Visualize the building footprints using the colors stored in the file.

When an OpenStreetMap file does not specify the color of a building, the table stores the color as `""`. Find the table rows that do not specify a building material and replace `""` with a dark gray color. Specify the color using a hexadecimal color code.

```
idxNoColor = buildingsLayer.Color == "";
buildingsLayer.Color(idxNoColor) = "#565656";
```

Find the unique color codes stored in the geospatial table.

```
colors = unique(buildingsLayer.Color)
```

```
colors = 5×1 string
    "#027880"
    "#3BAA32"
```

```
    "#565656"
    "#808000"
    "#8E4585"
```

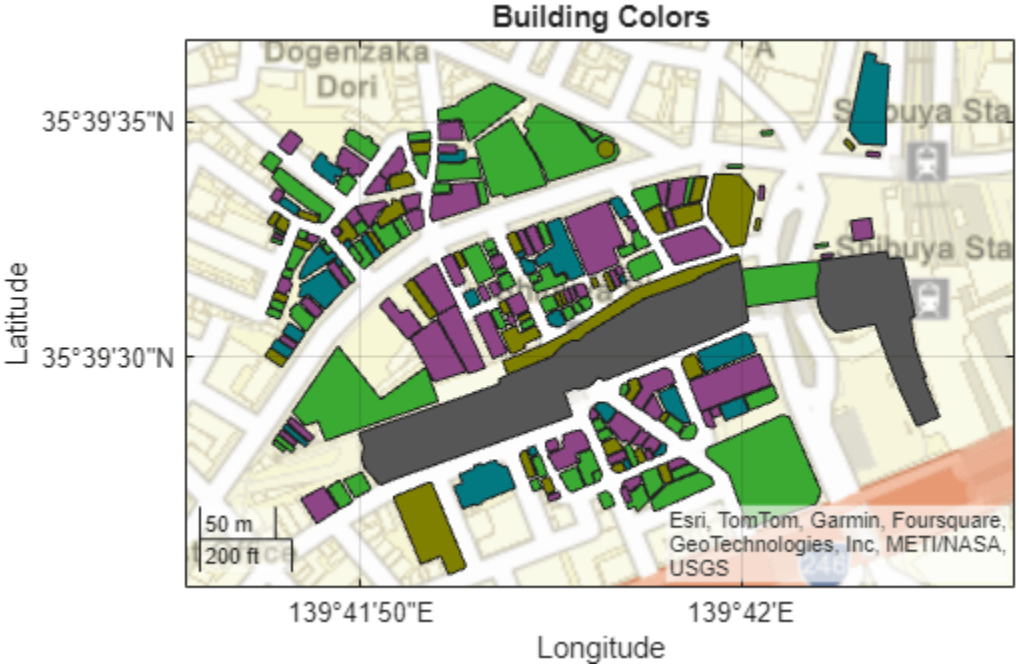
Display the building footprints on a road map. To apply each color, display the building footprints using a loop. Within the loop:

- Get a color code from the list of color codes.
- Find the table rows that use the color code.
- Create a subtable from the table rows.
- Display the buildings from the subtable. Specify the plot color using the color code.

```
figure
geobasemap streets
hold on

for k = 1:length(colors)
    c = colors(k);
    idxc = buildingsLayer.Color == c;
    rows = buildingsLayer(idxc,:);
    geoplot(rows,FaceColor=c);
end
```

Add a title and make the buildings more opaque.

```
title("Building Colors")
alpha(1)
```



Add Data Tips to Buildings

Display the names and heights of buildings by using data tips.

When an OpenStreetMap file does not specify the name of a building, the table stores the name as `""`. Find the table rows that do not specify a building name and replace `""` with `"unknown"`.

```
idxNoName = buildingsLayer.Name == "";
buildingsLayer.Name(idxNoName) = "unknown";
```

Display the buildings using polygons over a satellite basemap. Prepare to add data tips to the buildings by returning the `Polygon` object as `pg`. Make the buildings more opaque.

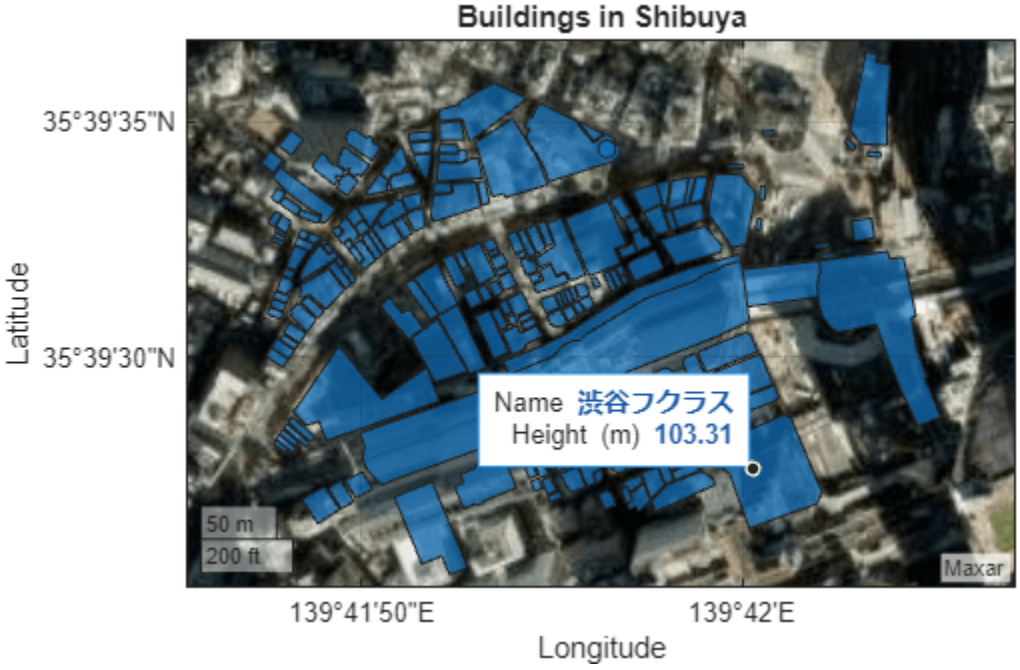
```
figure
geobasemap satellite
pg = geoplot(buildingsLayer);
title("Buildings in Shibuya")
alpha(0.7)
```

Customize the content of the data tips by adding the building names and heights to the data tip template of the `Polygon` object. For more information about creating custom data tips on polygon shapes, see “Add Data Tips to Point, Line, and Polygon Shapes” on page 3-97.

```
dtRowName = dataTipTextRow("Name",buildingsLayer.Name);
pg.DataTipTemplate.DataTipRows(1) = dtRowName;

dtRowMaterial = dataTipTextRow("Height (m)",buildingsLayer.MaxHeight);
pg.DataTipTemplate.DataTipRows(2) = dtRowMaterial;
```

Find the table row that corresponds to the tallest building. Then, add a data tip to the tallest building. Specify the `Polygon` object and the row number as input to the `datatip` function.

```
tallest = max(buildingsLayer.MaxHeight);
idxTallest = find(buildingsLayer.MaxHeight == tallest);

dt = datatip(pg,DataIndex=idxTallest);
```



[1] You can download OpenStreetMap files from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

See Also

Functions

`readgeotable` | `geoplot` | `geoscatter` | `datatip` | `dataTipTextRow`

Related Examples

- “Read Data from OpenStreetMap Files” on page 2-21
- “Create Custom Data Tips”

Export Vector Geodata

When you want to share geodata you are working with, Mapping Toolbox functions can export it two principal formats, shapefiles and KML files. Shapefiles are binary files that can contain point, line, vector, and polygon data plus attributes. Shapefiles are widely used to exchange data between different geographic information systems. KML files are text files that can contain the same type of data, and are used mainly to upload geodata the Web. The toolbox functions `shapewrite` and `kmlwrite` export to these formats.

To format attributes, `shapewrite` uses an auxiliary structure called a *DBF spec*, which you can generate with the `makedbfspec` function. Similarly, you can provide attributes to `kmlwrite` to format as a table by providing an *attribute spec*, a structure you can generate using the `makeattribspec` function or create manually.

For examples of and additional information about reading and writing shapefiles and DBF specs, see the documentation for `shapeinfo`, `shaperead`, `shapewrite`, and `makedbfspec`. For information about creating KML files, see “Export KML Files for Viewing in Earth Browsers” on page 2-34.

Export KML Files for Viewing in Earth Browsers

Keyhole Markup Language (KML) is an XML dialect for formatting 2-D and 3-D geodata for display in "Earth browsers," such as Google Earth™ mapping service, Google Maps™ mapping service, Google Mobile™ wireless service, and NASA WorldWind. Other Web browser applications, such as Yahoo!® Pipes, also support KML either by rendering or generating files. A KML file specifies a set of features (placemarks, images, polygons, 3-D models, textual descriptions, etc.) and how they are to be displayed in browsers and applications.

Each place must at least have an address or a longitude and a latitude. Places can also have textual descriptions, including hyperlinks. KML files can also specify display styles for markers, lines and polygons, and "camera view" parameters such as tilt, heading, and altitude. You can generate placemarks in KML files for individual points and sets of points that include attributes in table form. You can include HTML markups in these tables, with or without hyperlinks, but you cannot currently control the camera view of a placemark. (However, the users of an Earth browser can generally control their views of it).

Generate a Single Placemark Using kmlwritepoint

This example shows how to generate a placemark using `kmlwritepoint` by specifying the latitude and longitude that identifies a location. This example also specifies the icon used for the placemark and the text that appears in the balloon associated with the placemark.

```
lat =  42.299827;
lon = -71.350273;
description = sprintf('%s<br>%s</b><br>%s</b>', ...
    '3 Apple Hill Drive', 'Natick, MA. 01760', ...
    'https://www.mathworks.com');
name = 'The MathWorks, Inc.';
iconFilename = ...
    'https://www.mathworks.com/products/product_listing/images/ml_icon.gif';
iconScale = 1.0;
filename = 'MathWorks.kml';
kmlwritepoint(filename, lat, lon, ...
    'Description', description, 'Name', name, ...
    'Icon', iconFilename, 'IconScale', iconScale);
```

This code produces the following KML file.

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
   <Document>
      <name>MathWorks</name>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description>3 Apple Hill Drive<br>Natick, MA. 01760</b>;
                     <br>https://www.mathworks.com</b>;
         </description>
         <name>The MathWorks, Inc.</name>
         <Style>
            <IconStyle>
               <Icon>
                 <href>
                   https://www.mathworks.com/products/product_listing/images/ml_icon.gif
                 </href>
               </Icon>
               <scale>1</scale>
            </IconStyle>
         </Style>
         <Point>
            <coordinates>-71.350273,42.299827,0.0</coordinates>
         </Point>
      </Placemark>
   </Document>
</kml>
```

If you view this in an Earth Browser, notice that the text inside the placemark, "https://www.mathworks.com," was automatically rendered as a hyperlink. The Google Earth service also adds a link called "Directions". `kmlwritepoint` does not include location coordinates in placemarks. This is because it is easy for users to read out where a placemark is by mousing over it or by viewing its Properties dialog box.

Generate Placemarks from Addresses

This example shows how to generate a placemark using street addresses or more general addresses such as postal codes, city, state, or country names, instead of latitude and longitude information. If the viewing application is capable of looking up addresses, such placemarks can be displayed in appropriate, although possibly imprecise, locations. (Note that the Google Maps service does not support address-based placemarks.)

When you use addresses, `kmlwrite` creates an `<address>` element for each placemark rather than `<point>` elements containing `<coordinates>` elements. For example, here is code for `kmlwrite` that generates address-based placemarks for three cities in Australia from a cell array:

```
  address = {'Perth, Australia', ...
             'Melbourne, Australia', ...
             'Sydney, Australia'};
  filename = 'Australian_Cities.kml';
  kmlwrite(filename, address, 'Name', address);
```

The generated KML file has the following structure and content:

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
   <Document>
      <name>Australian_Cities</name>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Perth, Australia</name>
         <address>Perth, Australia</address>
      </Placemark>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Melbourne, Australia</name>
         <address>Melbourne, Australia</address>
      </Placemark>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Sydney, Australia</name>
         <address>Sydney, Australia</address>
      </Placemark>
   </Document>
</kml>
```

Export Point Geostructs to Placemarks

This example shows how to read data from shapefiles and generate a KML file that identifies all or selected attributes, which you can then view in an earth browser such as Google Earth. It also shows how to customize placemark icons and vary them according to attribute values.

Suppose you have a shapefile that contains a database of 162 tsunami events reported between 1950 and 2006, described as point locations with 21 variables (including 18 attributes). The steps below select some of these events from the shapefiles and displays them as tables in exported KML placemarks.

1. Read the tsunami shapefiles, selecting certain attributes.

   There are several ways to select attributes from shapefiles. One is to pass `shaperead` a cell array of attribute names in the Attributes parameter. For example, you might just want to map the maximum wave height, the suspected cause, and also show the year, location and country for each event. Set up a cell array with the corresponding attribute field names as follows, remembering that field names are case-sensitive.

   ```
   attrs = {'Max_Height','Cause','Year','Location','Country'};
   ```

   Since the data file uses latitude and longitude coordinates, you need to specify `'UseGeoCoords',true` to ensure that `shaperead` returns a geostruct (having `Lat` and `Lon` fields).

   ```
   tsunamis = shaperead('tsunamis.shp','useGeoCoords',true,...
                        'Attributes',attrs);
   ```

   Look at the first record in the `tsunamis` geostruct returned by `shaperead`.

   ```
   tsunamis(1)
   ```

   ```
              Geometry: 'Point'
              Lon: 128.3000
              Lat: -3.8000
       Max_Height: 2.8000
            Cause: 'Earthquake'
             Year: 1950
         Location: 'JAVA TRENCH, INDONESIA'
          Country: 'INDONESIA'
   ```

2. Export the tsunami data to a KML file with `kmlwrite`

   By default, `kmlwrite` outputs all attribute data in a geostruct to a KML formatted file as an HTML table containing unstyled text. When you view it, the Google Earth program supplies a default marker.

   ```
   kmlfilename = 'tsunami1.kml';
   kmlwritepoint(kmlfilename,tsunamis(1).Lat,tsunamis(1).Lon);
   ```

3. View the placemarks in an earth browser. For example, you can view KML files with the Google Earth browser, which must be installed on your computer.

   For Windows, use the `winopen` function:

   ```
   winopen(filename)
   ```

   For Linux, if the file name is a partial path, use the following commands:

   ```
   cmd = 'googleearth ';
   fullfilename = fullfile(pwd, filename);
   system([cmd fullfilename])
   ```

   For Mac, if the file name is a partial path, use the following commands:

```
cmd = 'open -a Google\ Earth '
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

4 Customize the placemark contents

To customize the HTML table in the placemark, use the `makeattribspec` function. Create an attribute spec for the `tsunamis` geostruct and inspect it.

```
attribspec = makeattribspec(tsunamis)

attribspec =
    Max_Height: [1x1 struct]
         Cause: [1x1 struct]
          Year: [1x1 struct]
      Location: [1x1 struct]
       Country: [1x1 struct]
```

Format the label for `Max_Height` as bold text, give units information about `Max_Height`, and also set the other attribute labels in bold.

```
attribspec.Max_Height.AttributeLabel = '<b>Maximum Height</b>';
attribspec.Max_Height.Format = '%.1f Meters';
attribspec.Cause.AttributeLabel = '<b>Cause</b>';
attribspec.Year.AttributeLabel = '<b>Year</b>';
attribspec.Year.Format = '%.0f';
attribspec.Location.AttributeLabel = '<b>Location</b>';
attribspec.Country.AttributeLabel = '<b>Country</b>';
```

When you use the attribute spec, all the attributes it lists are included in the placemarks generated by `kmlwrite` unless you remove them from the spec manually (e.g., with `rmfield`).

5 Customize the placemark icon

You can specify your own icon using `kmlwrite` to use instead of the default pushpin symbol. The black-and-white bullseye icon used here is specified as URL for an icon in the Google KML library.

```
iconname = ...
  'http://maps.google.com/mapfiles/kml/shapes/placemark_circle.png';
kmlwritepoint(kmlfilename,tsunamis(1).Lat,tsunamis(1).Lon, ...
  'Description',attribspec,'Name',{tsunamis(1).Location}, ...
  'Icon',iconname,'IconScale',2);
```

6 Vary placemark size by tsunami height

To vary the size of placemark icons, specify an icon file and a scaling factor for every observation as vectors of names (all the same) and scale factors (all computed individually) when writing a KML file. Scale the width and height of the markers to the log of `Max_Height`. Scaling factors for point icons are data-dependent and can take some experimenting with to get right.

```
% Create vector with log2 exponents of |Max_Height| values
[loghgtx loghgte] = log2([tsunamis.Max_Height]);
% Create a vector replicating the icon URL
iconnames = cellstr(repmat(iconname,numel(tsunamis),1));
kmlwritepoint(kmlfilename,tsunamis(1).Lat,tsunamis(1).Lon,
    'Description',attribspec,...
    'Name',{tsunamis(1).Location},'Icon',iconname,...
    'IconScale',loghgte);
```

See Also

Related Examples

- “Supported Geospatial File Formats for Import and Export” on page 2-2

Select Shapefile Data to Read

The `shaperead` function provides you with a powerful method, called a *selector*, to select only the data fields and items you want to import from shapefiles.

A selector is a cell array with two or more elements. The first element is a handle to a predicate function (a function with a single output argument of type `logical`). Each remaining element is a character vector indicating the name of an attribute.

For a given feature, `shaperead` supplies the values of the attributes listed to the predicate function to help determine whether to include the feature in its output. The feature is excluded if the predicate returns `false`. The converse is not necessarily true: a feature for which the predicate returns `true` may be excluded for other reasons when the selector is used in combination with the bounding box or record number options.

The following examples are arranged in order of increasing sophistication. Although they use MATLAB function handles, anonymous functions, and nested functions, you do not need to be familiar with these features to use the selectors for `shaperead`.

Example 1: Predicate Function in Separate File

1. Define the predicate function in a separate file. (Prior to Release 14, this was the only option available.) Create a file named `roadfilter.m`, with the following contents:

   ```
   function result = roadfilter(roadclass,roadlength)
   mininumClass = 4;
   minimumLength = 200;
   result = (roadclass  >= mininumClass) && ...
            (roadlength >= minimumLength);
   end
   ```

2. You can then call `shaperead` like this:

   ```
   roadselector = {@roadfilter, 'CLASS', 'LENGTH'}

   roadselector =
       @roadfilter    'CLASS'    'LENGTH'

   s = shaperead('concord_roads.shp', 'Selector', roadselector)

   s =
   115x1 struct array with fields:
       Geometry
       BoundingBox
       X
       Y
       STREETNAME
       RT_NUMBER
       CLASS
       ADMIN_TYPE
       LENGTH
   ```

   or, in a slightly more compact fashion, like this:

   ```
   s = shaperead('concord_roads.shp',...
                 'Selector', {@roadfilter, 'CLASS', 'LENGTH'})
   ```

```
s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Prior to Version 7 of the Mapping Toolbox software, putting the selector in a file or local function of its own was the only way to work with a selector.

Note that if the call to `shaperead` took place within a function, then `roadfilter` could be defined in a local function thereof rather than in a file of its own.

Example 2: Predicate as Function Handle

As a simple variation on the previous example, you could assign a function handle, `roadfilterfcn`, and use it in the selector:

```
roadfilterfcn = @roadfilter
s = shaperead('concord_roads.shp',...
              'Selector', {roadfilterfcn, 'CLASS', 'LENGTH'})
roadfilterfcn =
@roadfilter
s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Example 3: Predicate as Anonymous Function

Having to define predicate functions in files of their own, or even as local functions, may sometimes be awkward. Anonymous functions allow the predicate function to be defined right where it is needed. For example:

```
roadfilterfcn = ...
    @(roadclass, roadlength) (roadclass >= 4) && ...
    (roadlength >= 200)

roadfilterfcn =
    @(roadclass, roadlength) (roadclass >= 4) ...
               && (roadlength >= 200)

s = shaperead('concord_roads.shp','Selector', ...
```

```
              {roadfilterfcn, 'CLASS', 'LENGTH'})

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Example 4: Predicate (Anonymous Function) Defined Within Cell Array

There is actually no need to introduce a function handle variable when defining the predicate as an anonymous function. Instead, you can place the whole expression within the selector cell array itself, resulting in somewhat more compact code. This pattern is used in many examples throughout the Mapping Toolbox documentation and function help.

```
s = shaperead('concord_roads.shp', 'Selector', ...
    {@(roadclass, roadlength)...
    (roadclass >= 4) && (roadlength >= 200),...
    'CLASS', 'LENGTH'})

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Example 5: Parametrizing the Selector; Predicate as Nested Function

In the previous patterns, the predicate involves two hard-coded parameters (called `minimumClass` and `minimumLength` in `roadfilter.m`), as well as the `roadclass` and `roadlength` input variables. If you use any of these patterns in a program, you need to decide on minimum cut-off values for `roadclass` and `roadlength` at the time you write the program. But suppose that you wanted to wait and decide on parameters like `minimumClass` and `minimumLength` at run time?

Fortunately, nested functions provide the additional power that you need to do this; they allow you to utilize workspace variables in as parameters, rather than requiring that the parameters be hard-coded as constants within the predicate function. In the following example, the workspace variables `minimumClass` and `minimumLength` could have been assigned through a variety of computations whose results were unknown until run-time, yet their values can be made available within the predicate as long as it is defined as a nested function. In this example the nested function is wrapped in a file called `constructroadselector.m`, which returns a complete selector: a handle to the predicate (named `nestedroadfilter`) and the two attribute names:

```
function roadselector = ...
    constructroadselector(minimumClass, minimumLength)
roadselector = {@nestedroadfilter, 'CLASS', 'LENGTH'};
    function result = nestedroadfilter(roadclass, roadlength)
        result = (roadclass  >= minimumClass) && ...
                 (roadlength >= minimumLength);
    end
end
```

The following four lines show how to use `constructroadselector`:

```
minimumClass = 4;     % Could be run-time dependent
minimumLength = 200;  % Could be run-time dependent

roadselector = constructroadselector(...
    minimumClass, minimumLength);

s = shaperead('concord_roads.shp', 'Selector', roadselector)

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Exporting Vector Data to KML

This example shows how to structure geographic point, line, and polygon vector data and export it to a Keyhole Markup Language (KML) file. KML is an XML-based markup language designed for visualizing geographic data on Web-based maps or "Earth browsers", such as Google Earth™, Google Maps™, NASA WorldWind, and the Esri® ArcGIS™ Explorer.

The following functions write geographic data to a KML file:

- `kmlwritepoint` — Write geographic points to KML file
- `kmlwriteline` — Write geographic line to KML file
- `kmlwritepolygon` — Write geographic polygon to KML file
- `kmlwrite` — Write geographic data to KML file

Define an Output Folder for the KML Files

This example creates several KML files and uses the variable `kmlFolder` to denote their location. The value used here is determined by the output of the `tempdir` command, but you could easily customize this.

```
kmlFolder = tempdir;
```

Create a cell array of the KML file names used in this example in order to optionally remove them from your KML output folder when the example ends.

```
kmlFilenames = {};
```

Create a Function Handle to Open an Earth Browser

A KML file can be opened in a variety of "Earth browsers", Web maps, or an editor. You can customize the following anonymous function handle to open a KML file. Executing this function handle launches the Google Earth browser, which must be installed on your computer. You can use the application by assigning the variable `useApplication` to `true` in your workspace or assign it to `true` here.

```
useApplication = exist("useApplication","var") && useApplication;

if useApplication
    if ispc
        % On Windows(R) platforms display the KML file with:
        openKML = @(filename) winopen(filename);
    elseif ismac
        % On Mac platforms display the KML file with:
        cmd = "open -a Google\ Earth ";
        openKML = @(filename) system([cmd filename]);
    else
        % On Linux platforms display the KML file with:
        cmd = "googleearth ";
        openKML = @(filename) system([cmd filename]);
    end
else
    % No "Earth browser" is installed on the system.
    openKML = @(filename) disp("");
end
```

Example 1: Write Single Point to KML File

This example writes a single point to a KML file.

Get the latitude and longitude coordinates for Paderborn, Germany.

```
paderborn = geocode("Paderborn, Germany","city");
lat = paderborn.Shape.Latitude;
lon = paderborn.Shape.Longitude;
```

Use `kmlwritepoint` to write the point to a KML file.

```
filename = fullfile(kmlFolder,"Paderborn.kml");
kmlwritepoint(filename,lat,lon);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 2: Write Single Point to KML File with Icon and Description

This example writes a single point to a KML file. The placemark includes an icon and a description with HTML markup.

Specify the latitude and longitude coordinates of the MathWorks® headquarters in Natick, Massachusetts.

```
lat = 42.299827;
lon = -71.350273;
```

Create a description for the placemark. Include HTML tags in the description to add new lines for the address.

```
description = sprintf("%s<br>%s</br><br>%s</br>", ...
   "3 Apple Hill Drive", "Natick, MA. 01760", ...
   "https://www.mathworks.com");
```

Assign `iconFilename` to a GIF file on the local system's network.

```
iconDir = fullfile(matlabroot,"toolbox","matlab","icons");
iconFilename = fullfile(iconDir,"matlabicon.gif");
```

Assign the name for the placemark.

```
name = "The MathWorks, Inc.";
```

Use `kmlwritepoint` to write the point and associated data to the KML file.

```
filename = fullfile(kmlFolder,"MathWorks.kml");
kmlwritepoint(filename,lat,lon,Description=description,Name=name, ...
   Icon=iconFilename);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 3: Write Multiple Points to KML File

This example writes the locations of cities in France to a KML file, including the names of the cities, and removes the default description table.

Read world cities data from a shapefile into a geospatial table. Create a subtable containing the data for cities in France.

```
worldcities = readgeotable("worldcities.shp");
n = ["Paris","Lyon","Nantes","Bordeaux","Marseille"];
cities = geocode(n,worldcities);
```

Write the data in the subtable to a KML file by using `kmlwrite`. Assign the name of the placemark to the name of the city. Remove the default description since the data has only one attribute.

```
filename = fullfile(kmlFolder,"French_Cities.kml");
kmlwrite(filename,cities,Name=cities.Name,Description={});
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 4: Write Multiple Points to KML File with Modified Attribute Table

This example writes placemarks at the locations of tsunami events, reported over several decades and tagged geographically by source location, to a KML file.

Read tsunamis data from a shapefile into a geospatial table.

```
tsunamis = readgeotable("tsunamis.shp",CoordinateSystemType="geographic");
```

Create an attribute specification.

```
attribspec = makeattribspec(tsunamis);
```

Remove all attributes from the specification, except for the attributes describing the maximum height, cause, year, location, and country.

```
desiredAttributes = ["Max_Height","Cause","Year","Location","Country"];
allAttributes = fieldnames(attribspec);
attributes = setdiff(allAttributes,desiredAttributes);
attribspec = rmfield(attribspec,attributes)
```

```
attribspec = struct with fields:
          Year: [1×1 struct]
         Cause: [1×1 struct]
       Country: [1×1 struct]
      Location: [1×1 struct]
    Max_Height: [1×1 struct]
```

Rename the `Max_Height` attribute to `Maximum Height` and highlight each attribute label in bold font.

```
attribspec.Max_Height.AttributeLabel = "<b>Maximum Height</b>";
attribspec.Cause.AttributeLabel = "<b>Cause</b>";
attribspec.Year.AttributeLabel = "<b>Year</b>";
attribspec.Location.AttributeLabel = "<b>Location</b>";
attribspec.Country.AttributeLabel = "<b>Country</b>";
```

Add `Meters` to the format of the maximum height attribute. Set the format of the year attribute to include no decimal places.

```
attribspec.Max_Height.Format = "%.1f Meters";
attribspec.Year.Format = "%.0f";
```

Export the selected attributes and tsunami source locations to a KML file by using the `kmlwrite` function.

```
filename = fullfile(kmlFolder,"Tsunami_Events.kml");
name = tsunamis.Location;
kmlwrite(filename,tsunamis,Description=attribspec,Name=name)
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 5: Write Single Point with a LookAt Virtual Camera to KML File

This example writes a single point with a LookAt virtual camera near Machu Picchu, Peru

Use a `geopoint` vector to define a LookAt virtual camera.

```
lat = -13.163111;
lon = -72.544945;
lookAt = geopoint(lat,lon);
lookAt.Range = 1500;
lookAt.Heading = 260;
lookAt.Tilt = 67;
```

Use `kmlwritepoint` to write the point location and LookAt information.

```
filename = fullfile(kmlFolder,"Machu_Picchu.kml");
alt = 2430;
name = "Machu Picchu";
kmlwritepoint(filename,lat,lon,alt,LookAt=lookAt,Name=name);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 6: Write Single Point with a Camera to KML File

This example writes a single point with a camera view of the Washington Monument in Washington D.C to a KML file. The marker is placed at the ground location of the camera.

Construct the camera.

```
camlat = 38.889301;
camlon = -77.039731;
camera = geopoint(camlat,camlon);
camera.Altitude = 500;
camera.Heading = 90;
camera.Tilt = 45;
camera.Roll = 0;
```

Use `kmlwritepoint` to write the point location and Camera information.

```
name = "Camera ground location";
lat = camera.Latitude;
lon = camera.Longitude;
filename = fullfile(kmlFolder,"WashingtonMonument.kml");
kmlwritepoint(filename,lat,lon,Camera=camera,Name=name)
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 7: Write Address Data to KML File

This example writes unstructured address data to a KML file.

Specify the names of several places of interest in the Boston area.

```
names = ["Boston", ...
    "Massachusetts Institute of Technology", ...
    "Harvard University", ...
    "Fenway Park", ...
    "North End"];
```

Specify the addresses of the places of interest.

```
addresses = [ ...
    "Boston, MA", ...
    "77 Massachusetts Ave, Cambridge, MA 02139", ...
    "Massachusetts Hall, Cambridge MA 02138", ...
    "4 Yawkey Way, Boston, MA", ...
    "134 Salem St, Boston, MA"];
```

Use `kmlwrite` to write the cell array of addresses to the KML file.

```
filename = fullfile(kmlFolder,"Places_of_Interest.kml");
kmlwrite(filename,addresses,Name=names,IconScale=1.5);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 8: Write Single Line to KML File

This example writes a single line connecting the top of Mount Washington to the Mount Washington Hotel in Carroll, New Hampshire, to a KML file.

Assign coordinate values for the area of interest.

```
lat_Mount_Washington = 44.270489039;
lon_Mount_Washington = -71.303246453;

lat_Mount_Washington_Hotel = 44.258056;
lon_Mount_Washington_Hotel = -71.440278;

lat = [lat_Mount_Washington lat_Mount_Washington_Hotel];
lon = [lon_Mount_Washington lon_Mount_Washington_Hotel];
```

Set the altitude to 6 feet, for the approximate height of a person.

```
alt = 6 * unitsratio("meters","feet");
```

Add a camera viewpoint from the Mount Washington Hotel.

```
clat = lat(2);
clon = lon(2);
camera = geopoint(clat,clon,Altitude=2,Tilt=90,Roll=0,Heading=90);
```

Use `kmlwriteline` to write the arrays to a KML file.

```
filename = fullfile(kmlFolder,"Mount_Washington.kml");
name = "Mount Washington";
kmlwriteline(filename,lat,lon,alt,Name=name,Color="k",Width=3, ...
    Camera=camera,AltitudeMode="relativeToGround");
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 9: Write GPS Track Log to KML File

This example writes a GPS track log to a KML file.

Read the track points log from the GPX file. The data in the log was obtained from a GPS wristwatch held while gliding over Mount Mansfield in Vermont, USA, on August 28, 2010.

```
track = readgeotable("sample_mixed.gpx",Layer="track_points");
```

Write the log to a KML file by using the `kmlwriteline` function. The elevation values obtained by the GPS are relative to sea level.

```
filename = fullfile(kmlFolder,"GPS_Track_Log.kml");
lat = track.Shape.Latitude;
lon = track.Shape.Longitude;
alt = track.Elevation;
name = "GPS Track Log";
kmlwriteline(filename,lat,lon,alt,Name=name,Color="k",Width=2, ...
    AltitudeMode="relativeToSeaLevel");
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 10: Write Circles to KML File

This example writes circles as lines around London City Airport to a KML file. The example includes a `LookAt` virtual camera.

Assign latitude and longitude values for the center of the feature.

```
lat0 = 51.50487;
lon0 = 0.05235;
```

Assign `azimuth` to [] to compute a complete small circle. Use the WGS84 ellipsoid.

```
azimuth = [];
spheroid = wgs84Ellipsoid;
```

Compute small circles of 3000, 2000, and 1000 meter radius. Assign a color value of blue, green, and red for each circle. Assign an elevation value of 100 meters (above ground) for each circle. Use a line `geoshape` vector to contain the data.

```
radius = 3000:-1000:1000;
colors = ["blue","green","red"];
elevation = 100;
circles = geoshape(0,0,Name="",Color="",Elevation=elevation);
for k = 1:length(radius)
    [lat,lon] = scircle1(lat0,lon0,radius(k),azimuth,spheroid);
    circles(k).Latitude = lat;
    circles(k).Longitude = lon;
    circles(k).Name = num2str(radius(k)) + " Meters";
    circles(k).Color = colors(k);
    circles(k).Elevation = elevation;
end
```

Use a `geopoint` vector to define a LookAt virtual camera with a viewpoint from the east of the airport and aligned with the runway.

```
lat = 51.503169;
lon =  0.105478;
range = 3500;
heading = 270;
tilt =  60;
lookAt = geopoint(lat,lon,Range=range,Heading=heading,Tilt=tilt);
```

Use `kmlwrite` to write the `geoshape` vector containing the circles and associated data to a KML file.

```
filename = fullfile(kmlFolder,"Small_Circles.kml");
kmlwrite(filename,circles,AltitudeMode="relativeToGround",Width=2, ...
    Name=circles.Name,Color=circles.Color,LookAt=lookAt);
```

Open the KML file. Using Google Earth, the `LookAt` view point is set when clicking on either one of the `1000 Meters`, `2000 Meters`, or `3000 Meters` strings in the Places list.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 11: Write Circular Polygons to KML File

This example writes circular polygons around London City Airport to a KML file. It includes a `LookAt` virtual camera and uses the same data calculated in step 9.

Change the `Geometry` property value of the geoshape vector to `"polygon"`. The polygons are drawn in the same order as the geoshape vector and are indexed from largest to smallest radii, thus each polygon will be visible in the browser.

```
circles.Geometry = "polygon";
```

Change the elevation of each polygon.

```
circles.Elevation = 1000:1000:3000;
```

Use a `geopoint` vector to define a LookAt virtual camera with a viewpoint from the east of the airport, aligned with the runway, and with a view of all three polygons.

```
lat = 51.501587;
lon = 0.066147;
range = 13110;
heading = 270;
tilt = 60;
lookAt = geopoint(lat,lon,Range=range,Heading=heading,Tilt=tilt);
```

Use `kmlwrite` to write the polygon geoshape vector containing the circular polygons and associated data to a KML file. Extrude the polygons to the ground. Set the polygon edge color to black and assign a face alpha value to provide visibility inside the polygon.

```
filename = fullfile(kmlFolder,"Small_Circle_Polygons.kml");
name = circles.Name;
color = circles.Color;
kmlwrite(filename,circles,AltitudeMode="relativeToGround",Extrude=true, ...
    Name=name,FaceColor=color,EdgeColor="k",FaceAlpha=0.6,LookAt=lookAt);
```

Open the KML file. Using Google Earth, the `LookAt` view point is set when clicking on either one of the `1000 Meters`, `2000 Meters`, or `3000 Meters` strings in the Places list.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 12: Write Polygon Data from Shapefile to KML file

This example writes polygon data from a shapefile to a KML file. The polygon faces are set with a color appropriate for political regions. The polygon faces are set with an alpha value to provide visibility inside the polygon.

```
states = readgeotable("usastatelo.shp");
colors = polcmap(height(states));
name = states.Name;
filename = fullfile(kmlFolder,"usastatelo.kml");
kmlwrite(filename,states,Name=name,FaceColor=colors,FaceAlpha=0.6, ...
    EdgeColor="k")
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 13: Write Polygon Contours to KML File

This example contours a grid in a local coordinate system, returns the contours in a geographic system, and writes the polygon contours to a KML file.

Create a grid in a local system.

```
X = -150000:10000:150000;
Y =  0:10000:300000;
[xmesh, ymesh] = meshgrid(X/50000, (Y - 150000)/50000);
Z = 8 + peaks(xmesh, ymesh);
```

Define a local geodetic origin near Frankfurt, Germany and an ellipsoidal height.

```
lat0 = 50.108;
lon0 = 8.6732;
h0 = 100;
```

Define contour levels.

```
levels = 0:2:18;
```

Contour the grid and return the output in a polygon geoshape vector.

```
[~, contourPolygons] = geocontourxy(X,Y,Z,lat0,lon0,h0,LevelList=levels);
```

Output the contours to a KML file. Set the faces with an alpha value. Set `CutPolygons` to `false` since the altitude values are not uniform. Clamp the polygons to the ground.

```
colors = parula(length(contourPolygons));
filename = fullfile(kmlFolder,"Contour_Polygons.kml");
kmlwrite(filename,contourPolygons,FaceColor=colors,FaceAlpha=0.6, ...
    EdgeColor="k",CutPolygons=false,AltitudeMode="clampToGround")
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 14: Write Polygon with Inner Ring to KML File

This example constructs a polygon with an inner ring around the Eiffel Tower and writes the polygon to a KML file. The polygon's altitude is set to 500 meters above ground.

```
lat0 = 48.858288;
lon0 = 2.294548;
outerRadius = .02;
innerRadius = .01;
[lat1,lon1] = scircle1(lat0,lon0,outerRadius);
[lat2,lon2] = scircle1(lat0,lon0,innerRadius);
[lon2,lat2] = poly2ccw(lon2,lat2);
lat = [lat1; NaN; lat2];
lon = [lon1; NaN; lon2];
alt = 500;
filename = fullfile(kmlFolder,"EiffelTower.kml");
```

Export the polygon to a KML file. Set the edge color to black, the face color to cyan, and the face alpha value.

```
kmlwritepolygon(filename,lat,lon,alt,EdgeColor="k",FaceColor="c", ...
    FaceAlpha=0.5)
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Delete Generated KML Files

Optionally, delete the new KML files from your KML output folder.

```
if ~useApplication
    for k = 1:length(kmlFilenames)
        delete(kmlFilenames{k})
    end
end
```

See Also

`kmlwrite` | `kmlwritepolygon` | `kmlwriteline` | `kmlwritepoint`

More About

- “Supported Geospatial File Formats for Import and Export” on page 2-2

Export Images and Raster Grids to GeoTIFF

This example shows how to write data referenced to standard geographic and projected coordinate systems to GeoTIFF files, using `geotiffwrite`. The Tagged-Image File Format (TIFF) has emerged as a popular format to store raster data. The GeoTIFF specification defines a set of TIFF tags that describe "Cartographic" information associated with the TIFF raster data. Using these tags, geolocated imagery or raster grids with coordinates referenced to a Geographic Coordinate System (latitude and longitude) or a (planar) Projected Coordinate System can be stored in a GeoTIFF file.

Setup: Define a Data Folder and File Name Utility Function

This example creates several temporary GeoTIFF files and uses the variable `datadir` to denote their location. The value used here is determined by the output of the `tempdir` command, but you could easily customize this. The contents of `datadir` are deleted at the end of the example.

```
datadir = fullfile(tempdir, 'datadir');
if ~exist(datadir, 'dir')
   mkdir(datadir)
end
```

Define an anonymous function to prepend `datadir` to the input file name:

```
datafile = @(filename)fullfile(datadir, filename);
```

Example 1: Write an Image Referenced to Geographic Coordinates

Write an image referenced to WGS84 geographic coordinates to a GeoTIFF file. The original image (boston_ovr.jpg) is stored in JPEG format, with referencing information external to the image file, in the "world file" (boston_ovr.jgw). The image provides a low resolution "overview" of Boston, Massachusetts, and the surrounding area.

Read the image from the JPEG file.

```
basename = 'boston_ovr';
imagefile = [basename '.jpg'];
A1 = imread(imagefile);
```

Obtain a referencing object from the world file.

```
worldfile = getworldfilename(imagefile);
R1 = worldfileread(worldfile,'geographic',size(A1));
```

Write the image to a GeoTIFF file.

```
filename1 = datafile([basename '.tif']);
geotiffwrite(filename1,A1,R1)
```

Return information about the file as a `RasterInfo` object. Note that the value of `CoordinateReferenceSystem` is a geographic coordinate reference system object.

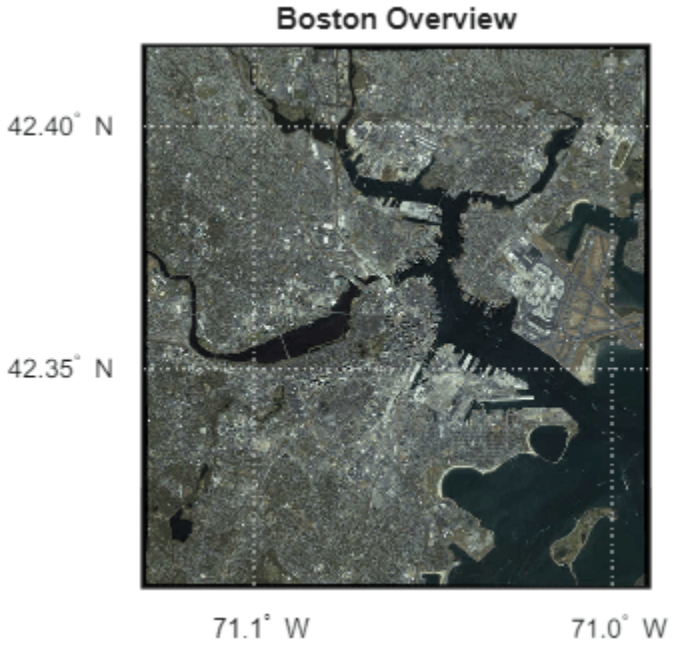
```
info1 = georasterinfo(filename1);
info1.CoordinateReferenceSystem
```

```
ans =
  geocrs with properties:
```

```
            Name: "WGS 84"
           Datum: "World Geodetic System 1984 ensemble"
        Spheroid: [1×1 referenceEllipsoid]
    PrimeMeridian: 0
        AngleUnit: "degree"
```

Re-import the new GeoTIFF file and display the Boston overview image, correctly located, on a map.

```
figure
usamap(R1.LatitudeLimits,R1.LongitudeLimits)
setm(gca,'PLabelLocation',0.05,'PlabelRound',-2,'PlineLocation',0.05)
geoshow(filename1)
title('Boston Overview')
```



Example 2: Write a Data Grid Referenced to Geographic Coordinates

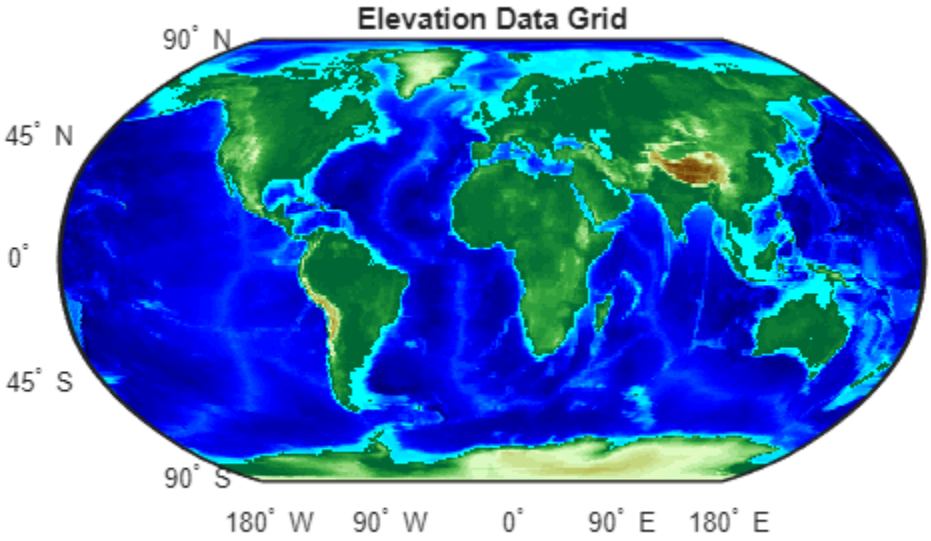
Load elevation raster data and a geographic cells reference object. Write the data grid to a GeoTIFF file.

```
load topo60c
Z2 = topo60c;
R2 = topo60cR;
filename2 = datafile('topo60c.tif');
geotiffwrite(filename2,Z2,R2)
```

The values in the data grid range from -7473 to 5731. Display the grid as a texture-mapped surface rather than as an intensity image.

```
figure
worldmap world
gridm off
setm(gca,'MLabelParallel',-90,'MLabelLocation',90)
tmap = geoshow(filename2,'DisplayType','texturemap');
```

```
demcmap(tmap.CData)
title('Elevation Data Grid')
```



Example 3: Change Data Organization of GeoTIFF Files

When you write data using `geotiffwrite` or read data using `readgeoraster`, the columns of the data grid start from north and the rows start from west. For example, the input data from `topo60c.mat` starts from south, but the output data from `topo60c.tif` starts from north.

```
R2.ColumnsStartFrom

ans =
'south'

[Z3,R3] = readgeoraster(filename2);
R3.ColumnsStartFrom

ans =
'north'
```

Therefore, the input data and data in the GeoTIFF file is flipped.

```
isequal(Z2,flipud(Z3))

ans = logical
   1
```

If you need the data in your workspace to match again, then flip the Z values and set the referencing object such that the columns start from the south:

```
R3.ColumnsStartFrom = 'south';
Z3 = flipud(Z3);
isequal(Z2,Z3)
```

```
ans = logical
   1
```

The data in the GeoTIFF file is encoded with positive scale values. Therefore, when you view the file with ordinary TIFF-viewing software, the northern edge of the data set is at the top. To make the data layout in the output file match the data layout of the input, you can construct a Tiff object and use it to reset some of the tags and the image data.

```
t = Tiff(filename2,'r+');

pixelScale = getTag(t,'ModelPixelScaleTag');
pixelScale(2) = -pixelScale(2);
setTag(t,'ModelPixelScaleTag',pixelScale);

tiepoint = getTag(t,'ModelTiepointTag');
tiepoint(5) = intrinsicToGeographic(R2,0.5,0.5);
setTag(t,'ModelTiepointTag',tiepoint);

setTag(t,'Compression', Tiff.Compression.None)

write(t,Z2);

rewriteDirectory(t)
close(t)
```

Verify the referencing object and data grid from the input data match the output data file. To do this, read the Tiff image and create a reference object. Then, compare the grids.

```
t = Tiff(filename2);
Atiff = read(t);
close(t)
Rtiff = georefcells(R2.LatitudeLimits,R2.LongitudeLimits,size(Atiff));

isequal(Z2,Atiff)

ans = logical
   1


isequal(R2,Rtiff)

ans = logical
   1
```

Example 4: Write an Image Referenced to a Projected Coordinate System

Write the Concord orthophotos to a single GeoTIFF file. The two adjacent (west-to-east) georeferenced grayscale (panchromatic) orthophotos cover part of Concord, Massachusetts, USA. The concord_ortho.txt file indicates that the data are referenced to the Massachusetts Mainland (NAD83) State Plane Projected Coordinate System. Units are meters. This corresponds to the GeoTIFF code number 26986 as noted in the GeoTIFF specification at http://geotiff.maptools.org/spec/geotiff6.html#6.3.3.1 under PCS_NAD83_Massachusetts.

Read the two orthophotos. Create reference objects for the orthophotos by reading their world files.

```
X_west = imread('concord_ortho_w.tif');
X_east = imread('concord_ortho_e.tif');
```

```
R_west = worldfileread("concord_ortho_w.tfw","planar",size(X_west));
R_east = worldfileread("concord_ortho_e.tfw","planar",size(X_east));
```

Merge the orthophotos.

```
[X4,R4] = mergetiles(X_west,R_west,X_east,R_east);
```

Write the data to a GeoTIFF file. Use the code number, 26986, indicating the PCS_NAD83_Massachusetts Projected Coordinate System.

```
coordRefSysCode = 26986;
filename4 = datafile('concord_ortho.tif');
geotiffwrite(filename4,X4,R4,'CoordRefSysCode',coordRefSysCode)
```

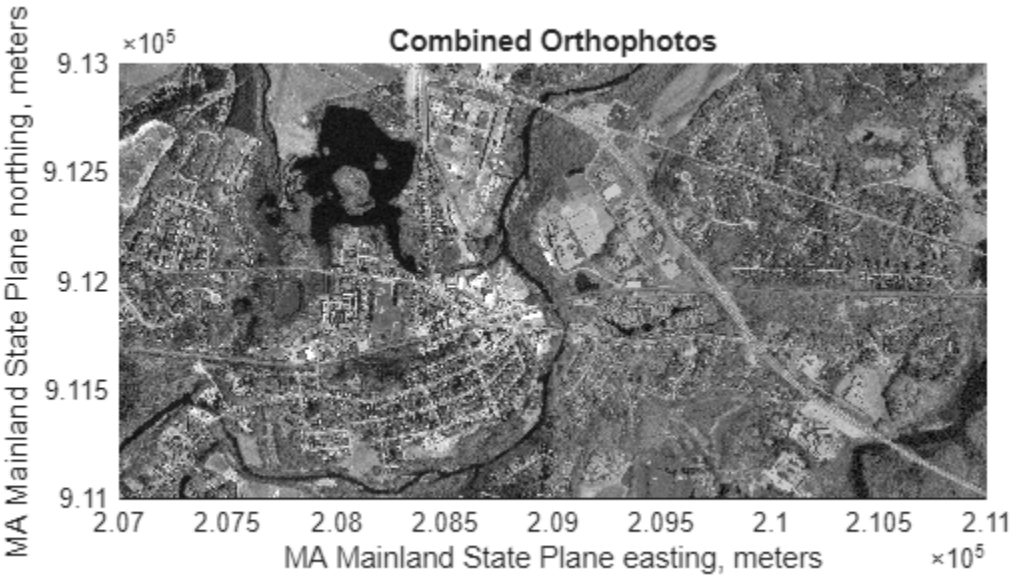
Return information about the file as a `RasterInfo` object. Note that the value of `CoordinateReferenceSystem` is a projected coordinate reference system object.

```
info4 = georasterinfo(filename4);
info4.CoordinateReferenceSystem
```

```
ans =
  projcrs with properties:

                    Name: "NAD83 / Massachusetts Mainland"
           GeographicCRS: [1×1 geocrs]
        ProjectionMethod: "Lambert Conic Conformal (2SP)"
              LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Display the combined Concord orthophotos.

```
figure
mapshow(filename4)
title('Combined Orthophotos')
xlabel('MA Mainland State Plane easting, meters')
ylabel('MA Mainland State Plane northing, meters')
```



Example 5: Write a Cropped Image from a GeoTIFF File

Write a subset of a GeoTIFF file to a new GeoTIFF file.

Read the RGB image and referencing information from the `boston.tif` GeoTIFF file.

```
[A5,R5] = readgeoraster('boston.tif');
```

Crop the image.

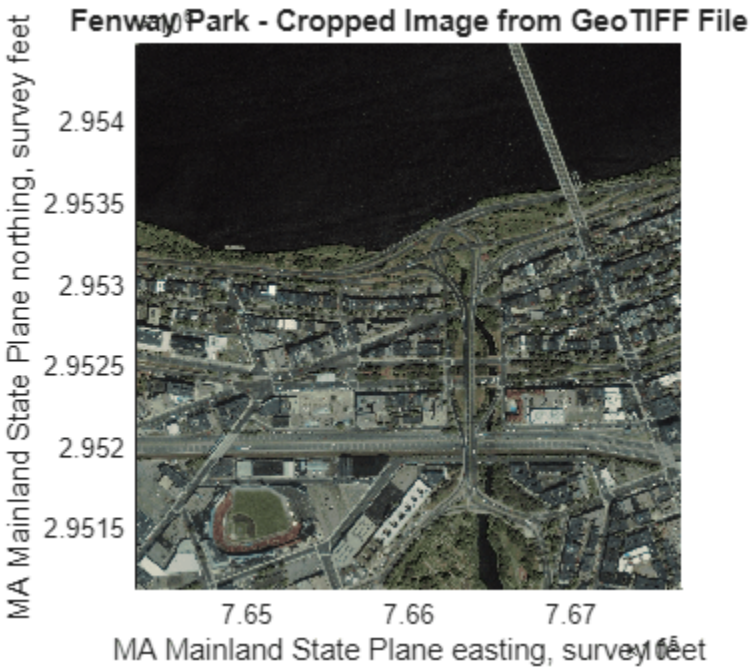
```
xlimits = [ 764318  767677];
ylimits = [2951122 2954482];
[A5crop,R5crop] = mapcrop(A5,R5,xlimits,ylimits);
```

Write the cropped image to a GeoTIFF file. Use the GeoKeyDirectoryTag from the original GeoTIFF file.

```
info5 = geotiffinfo('boston.tif');
filename5 = datafile('boston_subimage.tif');
geotiffwrite(filename5,A5crop,R5crop, ...
   'GeoKeyDirectoryTag',info5.GeoTIFFTags.GeoKeyDirectoryTag)
```

Display the GeoTIFF file containing the cropped image.

```
figure
mapshow(filename5)
title('Fenway Park - Cropped Image from GeoTIFF File')
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```



Example 6: Write Elevation Data to GeoTIFF File

Write elevation data for an area around South Boulder Peak in Colorado to a GeoTIFF file.

```
elevFilename = 'n39_w106_3arc_v2.dt1';
```

Read the DEM from the file. To plot the data using `geoshow`, the data must be of type `single` or `double`. Specify the data type for the raster using the `'OutputType'` name-value pair.

```
[Z6,R6] = readgeoraster(elevFilename,'OutputType','double');
```

Create a structure to hold the GeoKeyDirectoryTag information.

```
key = struct( ...
    'GTModelTypeGeoKey',[], ...
    'GTRasterTypeGeoKey',[], ...
    'GeographicTypeGeoKey',[]);
```

Indicate the data is in a geographic coordinate system by specifying the `GTModelTypeGeoKey` field as 2. Indicate that the reference object uses postings (rather than cells) by specifying the `GTRasterTypeGeoKey` field as 2. Indicate the data is referenced to a geographic coordinate reference system by specifying the `GeographicTypeGeoKey` field as 4326.

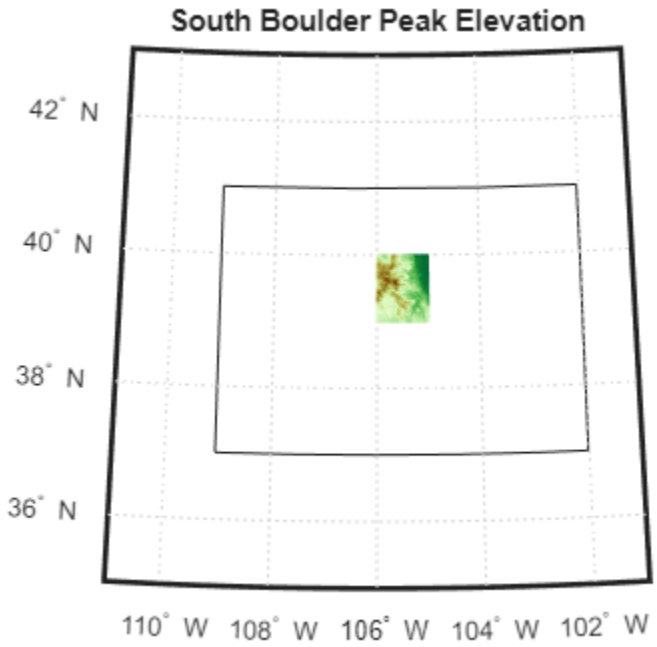
```
key.GTModelTypeGeoKey = 2;
key.GTRasterTypeGeoKey = 2;
key.GeographicTypeGeoKey = 4326;
```

Write the elevation data to a GeoTIFF file.

```
filename6 = datafile('southboulder.tif');
geotiffwrite(filename6,Z6,R6,'GeoKeyDirectoryTag',key)
```

Verify the data has been written to a file by displaying it. First, import vector data that represents the state boundary of Colorado using `readgeotable`. Then, display the boundary and GeoTIFF file.

```
GT = readgeotable('usastatelo.shp');
row = GT.Name == 'Colorado';
colorado = GT(row,:);

figure
usamap 'Colorado'
hold on
geoshow(colorado,'FaceColor','none')
g = geoshow(filename6,'DisplayType','mesh');
demcmap(g.ZData)
title('South Boulder Peak Elevation')
```



Example 7: Write Non-Image Data to a TIFF File

If you are working with a data grid that is class double with values that range outside the limits required of a floating point intensity image (0 <= data <= 1), and if you store the data in a TIFF file using `imwrite`, then your data will be truncated to the interval [0,1], scaled, and converted to uint8. Obviously it is possible for some or even all of the information in the original data to be lost. To avoid these problems, and preserve the numeric class and range of your data grid, use `geotiffwrite` to write the data.

Create sample Z data.

```
n = 512;
Z7 = peaks(n);
```

Create a referencing object to reference the rows and columns to X and Y.

```
R7 = maprasterref('RasterSize',[n n],'ColumnsStartFrom','north');
R7.XWorldLimits = R7.XIntrinsicLimits;
R7.YWorldLimits = R7.YIntrinsicLimits;
```

Create a structure to hold the GeoKeyDirectoryTag information. Set the model type to 1 indicating Projected Coordinate System (PCS).

```
key.GTModelTypeGeoKey = 1;
```

Set the raster type to 1 indicating PixelIsArea (cells).

```
key.GTRasterTypeGeoKey = 1;
```

Indicate a user-defined Projected Coordinate System by using a value of 32767.

```
key.ProjectedCSTypeGeoKey = 32767;
```

Write the data to a GeoTIFF file with `geotiffwrite`. For comparison, write a second file using `imwrite`.

```
filename_geotiff = datafile('zdata_geotiff.tif');
filename_tiff = datafile('zdata_tiff.tif');
geotiffwrite(filename_geotiff,Z7,R7,'GeoKeyDirectoryTag',key)
imwrite(Z7, filename_tiff);
```

When you read the file using `imread` the data values and class type are preserved. You can see that the data values in the TIFF file are not preserved.

```
geoZ  = imread(filename_geotiff);
tiffZ = imread(filename_tiff);
fprintf('Class type of Z: %s\n', class(Z7))
```

```
Class type of Z: double
```

```
fprintf('Class type of data in GeoTIFF file: %s\n', class(geoZ))
```

```
Class type of data in GeoTIFF file: double
```

```
fprintf('Class type of data in    TIFF file: %s\n', class(tiffZ))
```

```
Class type of data in    TIFF file: uint8
```

```
fprintf('Does data in GeoTIFF file equal Z: %d\n', isequal(geoZ, Z7))
```

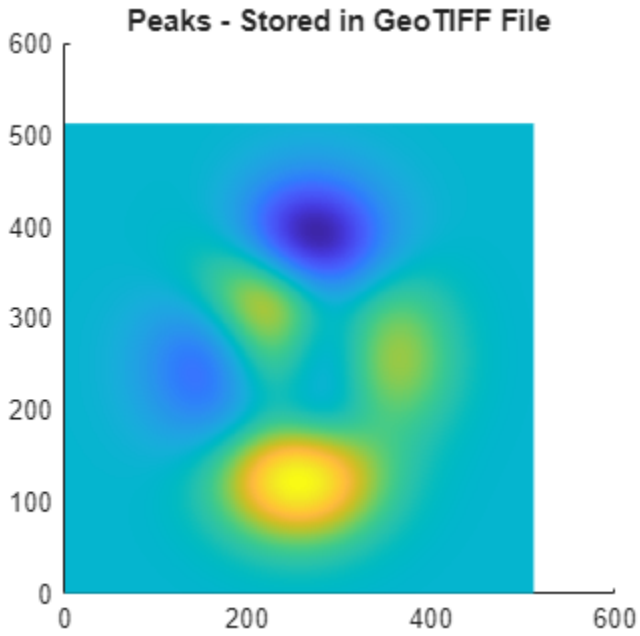
```
Does data in GeoTIFF file equal Z: 1
```

```
fprintf('Does data in    TIFF file equal Z: %d\n', isequal(tiffZ, Z7))
```

```
Does data in    TIFF file equal Z: 0
```

You can view the data grid using `mapshow`.

```
figure
mapshow(filename_geotiff,'DisplayType','texturemap')
title('Peaks - Stored in GeoTIFF File')
```



Example 8: Modify an Existing File While Preserving Meta Information

You may want to modify an existing file, but preserve most, if not all, of the meta information in the TIFF tags. This example converts the RGB image from the `boston.tif` file into an indexed image and writes the new data to an indexed GeoTIFF file. The TIFF meta-information, with the exception of the values of the BitDepth, BitsPerSample, and PhotometricInterpretation tags, is preserved.

Read the image from the `boston.tif` GeoTIFF file.

```
[A8,R8] = readgeoraster('boston.tif');
```

Use the MATLAB function, `rgb2ind`, to convert the RGB image to an indexed image X using minimum variance quantization.

```
[X8,cmap] = rgb2ind(A8,65536);
```

Obtain the TIFF tag information using `imfinfo`.

```
info8 = imfinfo('boston.tif');
```

Create a TIFF tags structure to preserve selected information from the `info` structure.

```
tags = struct( ...
    'Compression',  info8.Compression, ...
    'RowsPerStrip', info8.RowsPerStrip, ...
    'XResolution',  info8.XResolution, ...
    'YResolution',  info8.YResolution, ...
    'ImageDescription', info8.ImageDescription, ...
    'DateTime',     info8.DateTime, ...
    'Copyright',    info8.Copyright, ...
    'Orientation', info8.Orientation);
```

The values for the PlanarConfiguration and ResolutionUnit tags must be numeric rather than string valued, as returned by `imfinfo`. You can set these tags by using the constant properties from the Tiff class. For example, here are the possible values for the PlanarConfiguration constant property:

```
Tiff.PlanarConfiguration
```

```
ans = struct with fields:
      Chunky: 1
    Separate: 2
```

Use the string value from the `info` structure to obtain the desired value.

```
tags.PlanarConfiguration = ...
    Tiff.PlanarConfiguration.(info8.PlanarConfiguration);
```

Set the ResolutionUnit value in the same manner.

```
tags.ResolutionUnit = Tiff.ResolutionUnit.(info8.ResolutionUnit);
```

The Software tag is not set in the `boston.tif` file. However, `geotiffwrite` will set the `Software` tag by default. To preserve the information, set the value to the empty string which prevents the tag from being written to the file.

```
tags.Software = '';
```

Copy the GeoTIFF information from `boston.tif`.

```
geoinfo = geotiffinfo('boston.tif');
key = geoinfo.GeoTIFFTags.GeoKeyDirectoryTag;
```

Write to the GeoTIFF file.

```
filename8 = datafile('boston_indexed.tif');
geotiffwrite(filename8,X8,cmap,R8,'GeoKeyDirectoryTag',key,'TiffTags',tags)
```

View the indexed image.

```
figure
mapshow(filename8)
title('Boston - Indexed Image')
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```

Compare the information in the structures that should be equal by printing the values. The first column contains the field names, the second column contains the RGB information, and the third column contains the indexed information.

```
info_rgb = imfinfo('boston.tif');
info_indexed = imfinfo(filename8);
tagNames = fieldnames(tags);
tagNames(strcmpi('Software', tagNames)) = [];
names = [{'Height' 'Width'}, tagNames'];

spacing = 2;
fieldnameLength = max(cellfun(@length, names)) + spacing;
formatSpec = ['%-' int2str(fieldnameLength) 's'];

for k = 1:length(names)
    fprintf([formatSpec formatSpec formatSpec '\n'], ...
        names{k}, ...
        num2str(info_rgb.(names{k})), ...
        num2str(info_indexed.(names{k})))
end
```

```
Height              2881                2881
Width               4481                4481
Compression         Uncompressed        Uncompressed
RowsPerStrip        256                 256
XResolution         300                 300
YResolution         300                 300
ImageDescription    "GeoEye"            "GeoEye"
DateTime            2007:02:23 21:46:13 2007:02:23 21:46:13
Copyright           "(c) GeoEye"        "(c) GeoEye"
Orientation         1                   1
PlanarConfiguration Chunky              Chunky
ResolutionUnit      Inch                Inch
```

Compare the information that should be different, since you converted an RGB image to an indexed image, by printing the values. The first column contains the field names, the second column contains the RGB information, and the third column contains the indexed information.

```
names = {'FileSize', 'ColorType', 'BitDepth', ...
    'BitsPerSample', 'PhotometricInterpretation'};

fieldnameLength = max(cellfun(@length, names)) + spacing;
formatSpec = ['%-' int2str(fieldnameLength) 's'];
formatSpec2 = '%-17s';
for k = 1:length(names)
    fprintf([formatSpec formatSpec2 formatSpec2 '\n'], ...
        names{k}, ...
        num2str(info_rgb.(names{k})), ...
        num2str(info_indexed.(names{k})))
end
```

```
FileSize                  38729900         27925078
ColorType                 truecolor        indexed
BitDepth                  24               16
BitsPerSample             8  8  8          16
PhotometricInterpretation RGB              RGB Palette
```

Cleanup: Remove Data Folder

Remove the temporary folder and data files.

```
rmdir(datadir, 's')
```

Data Set Information

The files `boston.tif` and `boston_ovr.jpg` include materials copyrighted by GeoEye, all rights reserved. GeoEye was merged into the DigitalGlobe corporation on January 29th, 2013. For more information about the data sets, use the commands `type boston.txt` and `type boston_ovr.txt`.

The files `concord_orthow_w.tif` and `concord_ortho_e.tif` are derived using orthophoto tiles from the Bureau of Geographic Information (MassGIS), Commonwealth of Massachusetts, Executive Office of Technology and Security Services. For more information about the data sets, use the command `type concord_ortho.txt`. For an updated link to the data provided by MassGIS, see https://www.mass.gov/info-details/massgis-data-layers.

The DTED file `n39_w106_3arc_v2.dt1` is courtesy of the US Geological Survey.

See Also

`geotiffwrite` | `worldfileread` | `getworldfilename` | `geotiffinfo`

Create, Process, and Export Digital Surface Model from Lidar Data

This example shows how to process aerial lidar data received from an airborne lidar system into a GeoTIFF file. Import a LAZ file containing aerial lidar data, create a spatially referenced digital surface model (DSM) from the data, crop the DSM to an area of interest, and export the cropped DSM to a GeoTIFF file.

When you export a DSM to a GeoTIFF file, you also export the projected coordinate reference system (CRS) for the data. Projected CRSs associate *x*- and *y*-coordinates to locations on Earth. Specifying the projected CRS is important when creating a model because the same coordinates in different projected CRSs can refer to different locations.

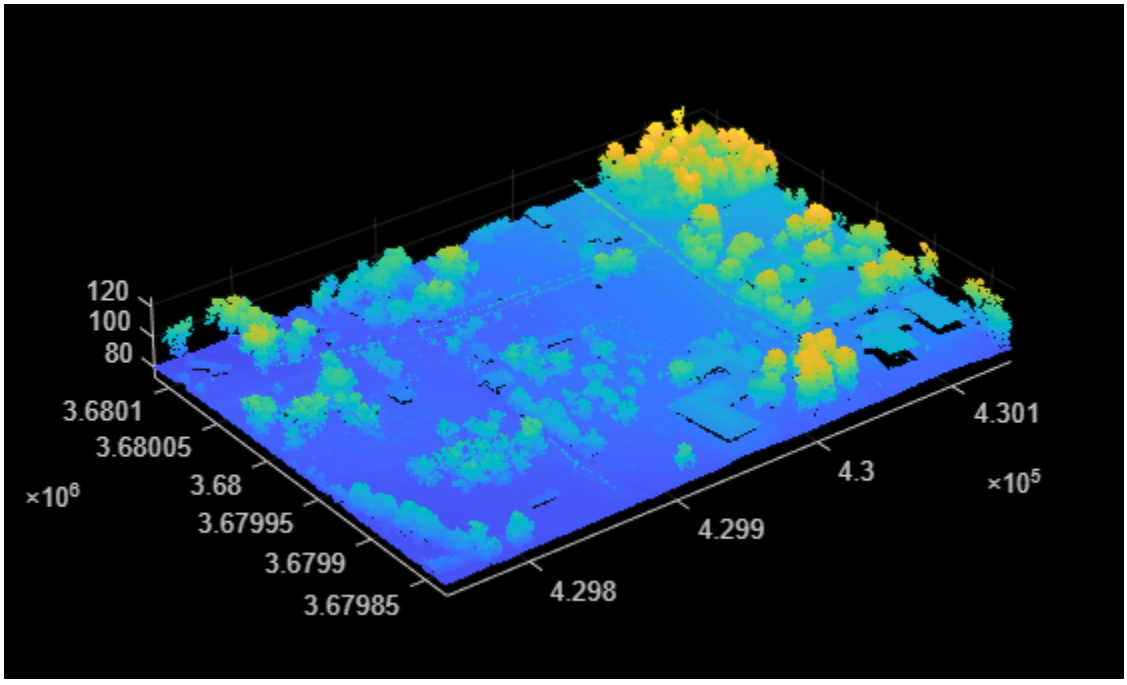
Read Aerial Lidar Data

Read 3-D point cloud data for an area near Tuscaloosa, Alabama from a LAZ file [1 on page 2-73]. The area includes roads, trees, and buildings.

```
lazFileName = fullfile(toolboxdir("lidar"),"lidardata","las","aerialLidarData.laz");
lasReader = lasFileReader(lazFileName);
ptCloud = readPointCloud(lasReader);
```

Display the data.

```
figure
pcshow(ptCloud.Location)
```



Create DSM

A DSM includes the elevations of ground points, the elevations of natural features such as trees, and the elevations of artificial features such as buildings. Create a DSM from the point cloud data by

using the `pc2dem` function. Use the maximum point from each element of the point cloud, which corresponds to the first return pulse of the lidar data, by specifying the `CornerFillMethod` as `"max"`. The function returns an array of elevation values and the *x*- and *y*-limits of the data.

```
gridRes = 1;
[Z,xlimits,ylimits] = pc2dem(ptCloud,gridRes,CornerFillMethod="max");
```

Spatially Reference DSM

Spatially reference the DSM by creating a map reference object.

```
R = maprefpostings(xlimits,ylimits,size(Z))
```

```
R =
  MapPostingsReference with properties:

            XWorldLimits: [429745.02 430146.02]
            YWorldLimits: [3679830.75 3680114.75]
              RasterSize: [285 402]
     RasterInterpretation: 'postings'
         ColumnsStartFrom: 'south'
            RowsStartFrom: 'west'
    SampleSpacingInWorldX: 1
    SampleSpacingInWorldY: 1
     RasterExtentInWorldX: 401
     RasterExtentInWorldY: 284
         XIntrinsicLimits: [1 402]
         YIntrinsicLimits: [1 285]
       TransformationType: 'rectilinear'
     CoordinateSystemType: 'planar'
             ProjectedCRS: []
```

The reference object contains information such as the limits, the distance between the points, and the directions of the columns and rows. By default, the reference object assumes that columns start from the south and rows start from the west. These default values are consistent with the output of the `pc2dem` function, which creates the elevation array such that the first element represents the southwesternmost point.

The `ProjectedCRS` property of the reference object is empty, which means the DSM is not associated with a projected CRS. Read the CRS from the LAZ file and update the `ProjectedCRS` property.

```
p = readCRS(lasReader);
R.ProjectedCRS = p;
disp(p)
```

```
  projcrs with properties:

                   Name: "NAD83 / UTM zone 16N"
          GeographicCRS: [1×1 geocrs]
       ProjectionMethod: "Transverse Mercator"
             LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

A projected CRS consists of a geographic CRS and several parameters that are used to transform coordinates to and from the geographic CRS. A geographic CRS consists of a datum (including a

reference ellipsoid), a prime meridian, and an angular unit of measurement. View the geographic CRS and its reference ellipsoid.

```
g = p.GeographicCRS
```

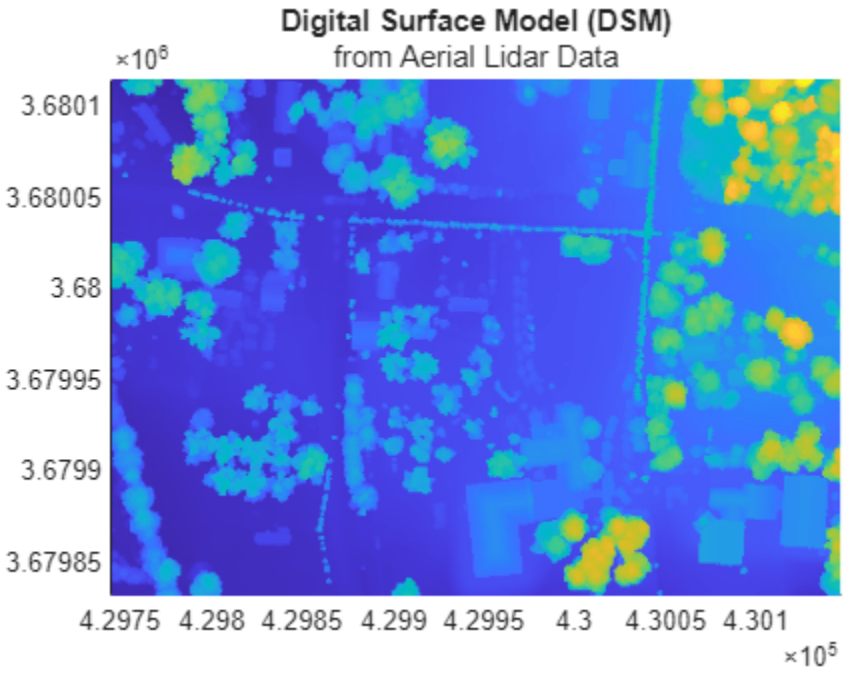
```
g =
  geocrs with properties:

            Name: "NAD83"
           Datum: "North American Datum 1983"
        Spheroid: [1×1 referenceEllipsoid]
    PrimeMeridian: 0
        AngleUnit: "degree"
```

```
g.Spheroid
```

```
ans =
referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'GRS 1980'
          LengthUnit: 'meter'
        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Display the spatially referenced DSM as an overhead surface by using the `mapshow` function.

```
figure
mapshow(Z,R,DisplayType="surface")
axis image
title("Digital Surface Model (DSM)","from Aerial Lidar Data")
```



Crop DSM to Area of Interest

Represent the DSM region as a polygon by using the `aoiquad` function. Specify a projected CRS that matches the CRS of the DSM.

```
bboxshape = aoiquad(xlimits,ylimits,p);
```

View the region using satellite imagery. You can visually confirm that the satellite imagery aligns with the DSM visualization created using the `mapshow` function.

```
regioncolors = lines(2);
geoplot(bboxshape, ...
    EdgeColor=regioncolors(1,:), ...
    FaceAlpha=0.2, ...
    LineWidth=2, ...
    DisplayName="Aerial Lidar Data Region")
hold on
geobasemap satellite
legend
```

Define and display an AOI in geographic coordinates. To use predefined points that correspond to roads on the east, north, and west, specify `interactivelySelectPoints` as `false`. Alternatively, you can interactively select four points by specifying `interactivelySelectPoints` as `true`. The `aoiquad` function defines the AOI by creating a quadrangle that bounds the points.

```
interactivelySelectPoints = false;
if interactivelySelectPoints
    [pointlat,pointlon] = ginput(4); %#ok<UNRCH>
else
    pointlat = [33.2571550 33.2551982 33.2551982 33.2571125];
    pointlon = [-87.7530648 -87.7530139 -87.7509086 -87.7509086];
end
aoigeo = aoiquad(pointlat,pointlon);

geoplot(aoigeo, ...
    EdgeColor=regioncolors(2,:), ...
    FaceAlpha=0.2, ...
    LineWidth=2, ...
    DisplayName="Selected Area of Interest")
```

Get the latitude and longitude limits of the AOI. Then, transform the coordinates to *x*- and *y*-limit coordinates. The geographic CRS underlying the `satellite` basemap is WGS84, while the geographic CRS underlying the DSM data is NAD83. NAD83 and WGS84 are similar, but not identical. As a result, there can be discrepancies in coordinates between the satellite imagery and DSM.

```
[aoilatlim,aoilonlim] = bounds(aoigeo);
[projx,projy] = projfwd(p,aoilatlim,aoilonlim);
```

Create an AOI in projected coordinates. Then, create the crop limits by finding the bounds of the *x*- and *y*-coordinates.

```
aoiproj = aoiquad(projx,projy,p);
[cropxlimits,cropylimits] = bounds(aoiproj);
```

Create a new spatially referenced DSM that contains data within the AOI.

```
[Zcrop,Rcrop] = mapcrop(Z,R,cropxlimits,cropylimits);
```

Export DSM to GeoTIFF File

Write the cropped DSM to a GeoTIFF file called `lidardsm.tif`. Specify the projected CRS by using the `CoordRefSysCode` argument. The metadata for the LAZ file [1 on page 2-73] indicates the projected CRS is UTM Zone 16N, specified by EPSG authority code 26916.

```
datafile = "lidardsm.tif";
epsgCode = 26916;
geotiffwrite(datafile,Zcrop,Rcrop,CoordRefSysCode=epsgCode)
```

You can also find the authority code by displaying the well-known text (WKT) string for the projected CRS. For this WKT, the authority code is in the last line.

```
wktstring(p,Format="formatted")
```

```
ans =
    "PROJCRS["NAD83 / UTM zone 16N",
         BASEGEOGCRS["NAD83",
             DATUM["North American Datum 1983",
                 ELLIPSOID["GRS 1980",6378137,298.257222101,
                     LENGTHUNIT["metre",1]]],
             PRIMEM["Greenwich",0,
                 ANGLEUNIT["degree",0.0174532925199433]],
             ID["EPSG",4269]],
         CONVERSION["UTM zone 16N",
             METHOD["Transverse Mercator",
                 ID["EPSG",9807]],
             PARAMETER["Latitude of natural origin",0,
                 ANGLEUNIT["degree",0.0174532925199433],
                 ID["EPSG",8801]],
             PARAMETER["Longitude of natural origin",-87,
                 ANGLEUNIT["degree",0.0174532925199433],
                 ID["EPSG",8802]],
             PARAMETER["Scale factor at natural origin",0.9996,
                 SCALEUNIT["unity",1],
                 ID["EPSG",8805]],
             PARAMETER["False easting",500000,
                 LENGTHUNIT["metre",1],
                 ID["EPSG",8806]],
             PARAMETER["False northing",0,
                 LENGTHUNIT["metre",1],
                 ID["EPSG",8807]]],
         CS[Cartesian,2],
             AXIS["easting",east,
                 ORDER[1],
                 LENGTHUNIT["metre",1]],
             AXIS["northing",north,
                 ORDER[2],
                 LENGTHUNIT["metre",1]],
         ID["EPSG",26916]]"
```

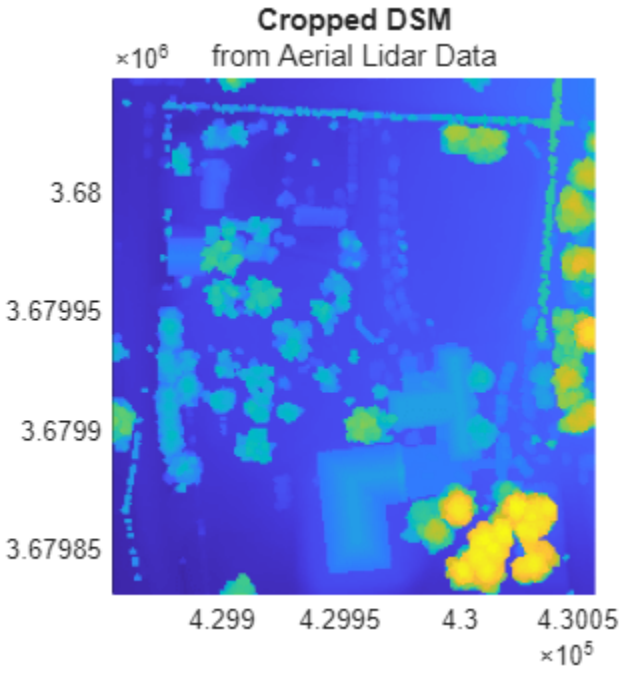
One way to validate the GeoTIFF file is to return information about the file as a `RasterInfo` object. For example, verify that the projected CRS is in the file by querying the `CoordinateReferenceSystem` property of the `RasterInfo` object.

```
info = georasterinfo(datafile);
info.CoordinateReferenceSystem
```

```
ans =
  projcrs with properties:

                   Name: "NAD83 / UTM zone 16N"
          GeographicCRS: [1×1 geocrs]
       ProjectionMethod: "Transverse Mercator"
             LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Another way to validate the GeoTIFF file is by displaying it. Read the new DSM as an array and a reference object by using the `readgeoraster` function. Then, display the DSM.

```
[Z2,R2] = readgeoraster(datafile);

figure
mapshow(Z2,R2,DisplayType="surface")
axis image
title("Cropped DSM","from Aerial Lidar Data")
```



You can use the GeoTIFF file in other applications that import GIS data. For example, RoadRunner enables you to add elevation data from GeoTIFF files to scenes.

References

[1] OpenTopography. “Tuscaloosa, AL: Seasonal Inundation Dynamics And Invertebrate Communities,” 2011. https://doi.org/10.5069/G9SF2T3K.

See Also

Functions

`geotiffwrite` | `mapshow` | `pc2dem` | `segmentGroundSMRF`

Objects

`MapPostingsReference` | `projcrs` | `lasFileReader`

Visualize Sea Ice Concentrations from GRIB Data

Visualize sea ice concentrations for an arctic region using data stored in the GRIB file format. GRIB files typically store meteorological, hydrological, or oceanographic data. In many cases, GRIB files store data using multiple bands. This example shows you how to:

- View sea ice concentrations for a single year.
- Animate sea ice concentrations over time.
- Compare sea ice concentrations for multiple years using histograms.

This example uses data from a GRIB file containing sea ice concentrations for 2010, 2012, 2014, 2016, 2018, 2020, and 2022 [1] on page 2-79[2] on page 2-79. The file stores the data as a grid of posting point samples referenced to geographic coordinates.

Visualize Concentrations for Single Year

Visualize sea ice concentrations for 2010 by using an `axesm`-based map.

Get Information About GRIB File

Get information about the GRIB file by creating a `RasterInfo` object. Extract the latitude limits, longitude limits, reference times, and number of bands from the object. Each data band corresponds to a year.

```
info = georasterinfo("seaice.grib");
latlim = info.RasterReference.LatitudeLimits;
lonlim = info.RasterReference.LongitudeLimits;
refTime = info.Metadata.ReferenceTime;
numBands = info.NumBands;
```

Read Data Band

Read the data band for 2010 by using the metadata stored in the `RasterInfo` object. `RasterInfo` objects store GRIB metadata using a table, where each row of the table corresponds to a data band.

Find the band that contains data for 2010.

```
yrs = year(info.Metadata.ReferenceTime);
band2010 = find(yrs == 2010);
```

Extract the band for 2010. Convert the concentrations to percentages.

```
[iceConcentration2010,R2010] = readgeoraster("seaice.grib",Bands=band2010);
percentIce2010 = 100*iceConcentration2010;
```

Get the reference time for the band.

```
refTime2010 = refTime(band2010);
refTime2010 = string(refTime2010,"MMMM d, yyyy");
```

Display Data Band

Load an ice colormap that is appropriate for ice concentration data. The colormap starts at dark blue and transitions to light blue and white.

```
load iceColormap.mat
```

Specify ice concentration levels from 0% to 100%. The level increases every 5%.

```
levels = 0:5:100;
```

Set up an `axesm`-based map for an arctic region by using the `setUpArcticAxesm` on page 2-79 helper function. The function creates a map in a polar stereographic projection, zooms into the arctic region, sets the colormap, and adds a labeled color bar. Specify the colormap as the ice colormap. Specify the tick labels for the color bar using the ice concentration levels.

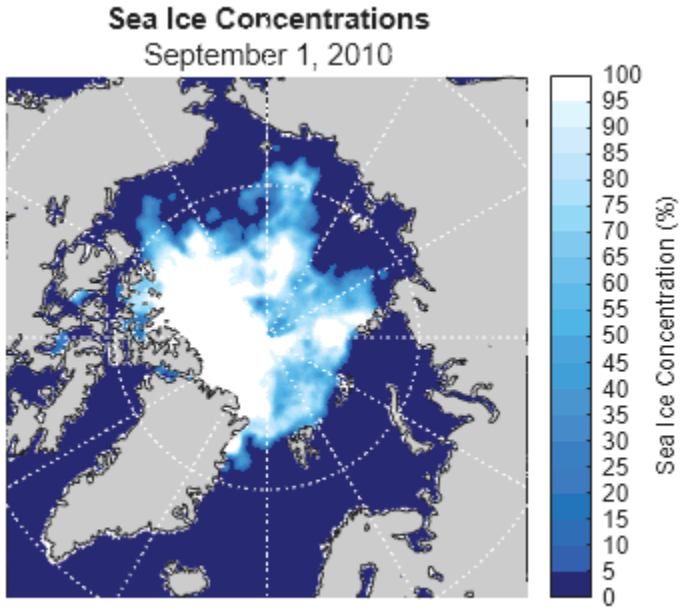
```
figure
setUpArcticAxesm(iceColormap,levels)
```

Provide geographic context for the map by displaying world land areas in gray. Then, display the sea ice concentrations as a surface. To make the surface appear under the land areas, display a matrix of zeros and apply color using the ice concentrations.

```
land = readgeotable("landareas.shp");
landColor = [0.8 0.8 0.8];
geoshow(land,FaceColor=landColor)

geoshow(zeros(R2010.RasterSize),R2010,DisplayType="surface",CData=percentIce2010)
```

Add a title and subtitle.

```
title("Sea Ice Concentrations")
subtitle(refTime2010)
```



Animate Concentrations Over Time

Visualize sea ice concentrations over time by using an animated GIF.

Read Data for All Bands

Read all the data bands into the workspace. Convert the concentrations to percentages.

```
[iceConcentration,R] = readgeoraster("seaice.grib");
percentIce = 100 * iceConcentration;
```

Create Animated GIF

Create an animated GIF and save it to a file.

Set up a new `axesm`-based map by using the `setUpArcticAxesm` on page 2-79 helper function. Avoid displaying the animation by using an invisible figure. Add a title.

```
f = figure(Color="w",Visible="off");
setUpArcticAxesm(iceColormap,levels)
title("Animation of Sea Ice Concentration")
```

Display the ice concentrations for 2010 as a surface. To use the same figure for each frame of the GIF, return the surface as a variable. This strategy enables you to display ice concentration data for different years by changing the color data for the surface. Then, provide geographic context for the map by displaying the land areas.

```
iceSurface = geoshow(zeros(R.RasterSize),R2010,DisplayType="surface",CData=percentIce2010);
geoshow(land,FaceColor=landColor)
```

Specify a name for the GIF. If a file with that name already exists, delete it.

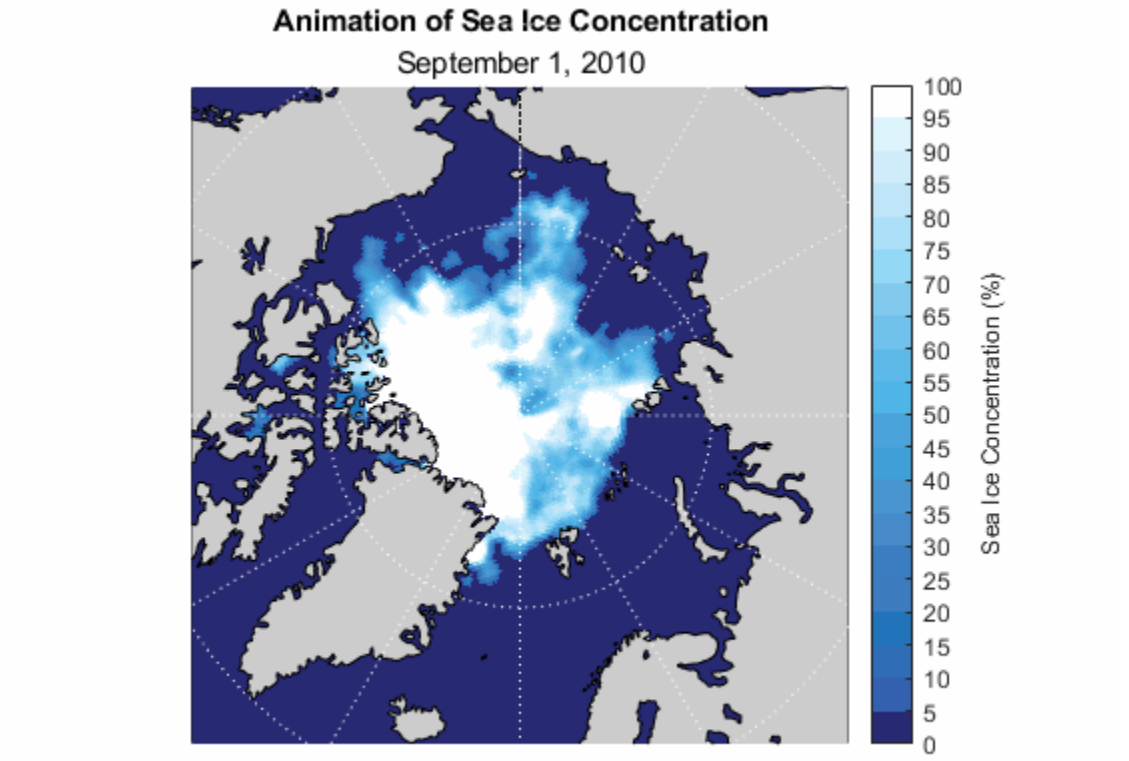
```
gifFilename = "seaice_animation.gif";
if exist(gifFilename,"file")
    delete(gifFilename)
end
```

Create an animation. For each year of data:

- Extract the data band for the year. Change the color data for the surface using the data band.
- Change the subtitle of the map to match the year.
- Write the figure to the GIF by using the `writeAnimatedGIF` on page 2-80 helper function.

```
for yr = 1:numBands
    % extract data band
    percentIceYr = percentIce(:,:,yr);
    iceSurface.CData = percentIceYr;

    % change subtitle
    refTimeYr = string(refTime(yr),"MMMM d, yyyy");
    subtitle(refTimeYr)

    % append map to GIF file
    drawnow
    writeAnimatedGIF(f,"seaice_animation.gif")
end
```

This image shows the animated GIF.



Compare Years Using Histograms

Quantitatively visualize the data using histograms. Compare the data for two years by displaying two histograms in the same figure. Avoid analyzing ice concentrations below a specified level using a threshold.

Select two years to compare.

```
yr1 = 2010;
yr2 = 2022;
```

For each year, find the data band, read the data band, convert the concentrations to percentages, and get the reference time.

```
bandYr1 = find(yrs == yr1);
bandYr2 = find(yrs == yr2);

[iceConcentrationYr1,RYr1] = readgeoraster("seaice.grib",Bands=bandYr1);
[iceConcentrationYr2,RYr2] = readgeoraster("seaice.grib",Bands=bandYr2);

percentIceYr1 = 100*iceConcentrationYr1;
percentIceYr2 = 100*iceConcentrationYr2;

refTimeYr1 = string(refTime(bandYr1),"MMMM d, yyyy");
refTimeYr2 = string(refTime(bandYr2),"MMMM d, yyyy");
```

Select a threshold for the ice concentrations.

threshold = 30 ;

For each year, extract the data points that are above the threshold. Find the percentage of data points that are above the threshold, excluding the `NaN` values.

```
% yr1
idxAboveThresholdYr1 = percentIceYr1 > threshold;
aboveThresholdYr1 = percentIceYr1(idxAboveThresholdYr1);
percentAboveThresholdYr1 = 100*numel(aboveThresholdYr1)/sum(~isnan(percentIceYr1),"all");

% yr2
idxAboveThresholdYr2 = percentIceYr2 > threshold;
aboveThresholdYr2 = percentIceYr2(idxAboveThresholdYr2);
percentAboveThresholdYr2 = 100*numel(aboveThresholdYr2)/sum(~isnan(percentIceYr2),"all");
```

Display the ice concentrations using two histograms. Specify the bin edges using the levels. Display the threshold using a vertical line.

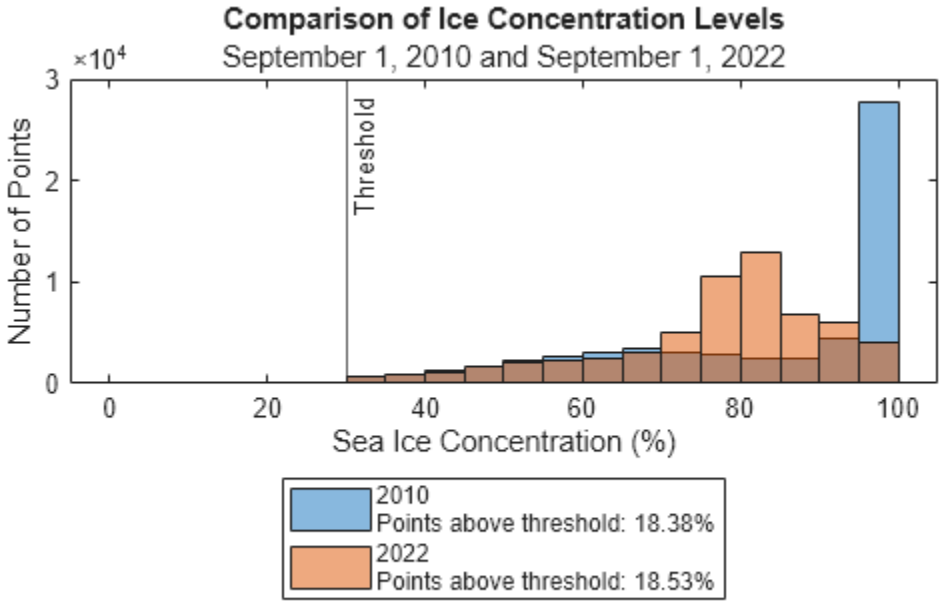
```
figure
histogram(aboveThresholdYr1,levels,FaceAlpha=0.5)
hold on
histogram(aboveThresholdYr2,levels,FaceAlpha=0.5)
xline(threshold,"-",{'Threshold '},HandleVisibility="off")
```

Display the year and the percentage of data points that are above the threshold in a legend.

```
labely1 = sprintf("%d \nPoints above threshold: %.2f%%",yr1,percentAboveThresholdYr1);
labely2 = sprintf("%d \nPoints above threshold: %.2f%%",yr2,percentAboveThresholdYr2);
legend(labely1,labely2,Location="southoutside")
```

Add titles and labels.

```
title("Comparison of Ice Concentration Levels")
subtitle(refTimeYr1 + " and " + refTimeYr2)
xlabel("Sea Ice Concentration (%)")
ylabel("Number of Points")
```



When data is stored as a grid of rectangular cells referenced to geographic coordinates, you can find the surface area by using the `areamat` function. However, most GRIB data is stored as grids of posting point samples. For more information about the difference between cells and posting points, see “Spatially Reference Imported Rasters” on page 6-41.

References

[1] Hersbach, H., B. Bell, P. Berrisford, G. Biavati, A. Horányi, J. Muñoz Sabater, J. Nicolas, et al. "ERA5 Hourly Data on Single Levels from 1940 to Present." Copernicus Climate Change Service (C3S) Climate Data Store (CDS), 2023. Accessed May 22, 2023. https://doi.org/10.24381/cds.adbb2d47.

[2] Neither the European Commission nor ECMWF is responsible for any use that may be made of the Copernicus information or data it contains.

Helper Functions

The `setupArcticAxesm` function sets up an `axesm`-based map for an arctic region using the colormap `cmapIn` and the ice concentration levels `levelsIn`. Within the function:

- Create an `axesm`-based map that uses a polar stereographic projection. Specify the origin using the coordinates of the North Pole.
- Zoom in to an arctic region by changing the *xy*-limits of the map. Specifying Cartesian limits instead of geographic limits enables you to display more data within the plot box.
- Set the colormap. Set the color limits using the minimum and maximum values of the ice concentration levels.

- Add a labeled color bar. Specify the ticks using the ice concentration levels.

```
function setUpArcticAxesm(cmapIn,levelsIn)
    % create axesm-based map
    abm = axesm("stereo",Origin=[90 0 0],Grid="on",GColor="w");

    % zoom in to arctic region
    nlim = 0.45;
    set(abm,XLim=[-nlim nlim],YLim=[-nlim nlim])

    % set colormap and color limits
    colormap(cmapIn)
    clim(abm,[min(levelsIn) max(levelsIn)])

    % add labeled color bar
    cb = colorbar;
    cb.Ticks = levelsIn;
    cb.Label.String = "Sea Ice Concentration (%)";
end
```

The `writeAnimatedGIF` function writes the figure `figIn` to the animated GIF with name `filename`. Within the function:

- Extract the color data from the figure. Convert the RGB image to an indexed image. Preserve the spatial resolution by turning off dithering.
- Write the image to the GIF. If the GIF does not exist, create a new GIF using the image. If the GIF does exist, append the image to the GIF.

```
function writeAnimatedGIF(figIn,filename)
    % create indexed image from figure
    data = getframe(figIn);
    RGB = data.cdata;
    [figFrame,figCmap] = rgb2ind(RGB,256,"nodither");

    % write image to GIF
    if ~exist(filename,"file")
        imwrite(figFrame,figCmap,filename,"gif",Loopcount=Inf,DelayTime=0.5)
    else
        imwrite(figFrame,figCmap,filename,"gif",WriteMode="append",DelayTime=0.5)
    end
end
```

See Also

Functions
`readgeoraster` | `georasterinfo` | `axesm`

Objects
`RasterInfo`

Related Examples

- “Supported Geospatial File Formats for Import and Export” on page 2-2

External Websites

- Climate Data Store Toolbox for MATLAB

Convert Coastline Data (GSHHG) to Shapefile Format

The Global Self-Consistent Hierarchical High-Resolution Geography (GSHHG) data set [1] on page 2-86 is a collection of geographic data that includes the boundaries of coastlines, lakes, and major rivers. GSHHG data is useful for creating base layers for maps and for performing analyses that require operations such as geographic searches and geometric quantities.

This example shows how to:

- Extract a subset of data from the GSHHG data set.
- Represent the data using polygon shapes.
- Merge the polygons that represent land areas and lakes.
- Save the merged data to a shapefile.

Index Coarse-Resolution GSHHG Data

Extract a file containing coarse GSHHG data from a GNU zipped file. Read the file into the workspace as a geographic data structure array.

```
files = gunzip("gshhs_c.b.gz");
filename = files{1};
```

Create an index file that accelerates future calls to the `gshhs` function. Note that, when you create an index file by using the `"createindex"` option, the `gshhs` function does not extract data.

```
indexfile = gshhs(filename,"createindex");
```

Read GSHHG Data

Specify the latitude and longitude limits of a region surrounding South America. Then, read GSHHG data for the region into the workspace as a geographic data structure array.

```
latlim = [-60  15];
lonlim = [-90 -30];
S = gshhs(filename,latlim,lonlim);
```

Convert the structure array into a geospatial table by using the `struct2geotable` function. A geospatial table is a `table` or `timetable` object that contains a `Shape` variable and attribute variables.

```
GT = struct2geotable(S);
```

Examine GSHHG Data

Note that the `Shape` variable contains a polygon shape in geographic coordinates.

```
GT.Shape
```

```
ans=155×1 geopolyshape array with properties:
              NumRegions: [155×1 double]
                NumHoles: [155×1 double]
                Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: []
```

GSHHS data includes multiple shoreline levels:

- Level 1 polygons represent land areas.
- Level 2 polygons represent lakes.
- Level 3 polygons represent islands within lakes.
- Level 4 polygons represent ponds within islands that are within lakes.

Find the shoreline levels in the South America GSHHG data by querying the `Level` variable of the geospatial table. The result indicates that the South America GSHHS data includes level 1, 2, and 3 polygons.

```
levels = GT.Level;
unique(levels)
```

```
ans = 3×1

     1
     2
     3
```

Alternatively, you can find the interpretation of the shoreline levels in the South America GSHHG data by querying the `LevelString` variable of the geospatial table. The result also indicates that the South America GSHHS data includes islands in lakes (level 3), lakes (level 2), and land (level 1).

```
levelstrings = GT.LevelString;
unique(levelstrings)
```

```
ans = 3×1 cell
    {'island_in_lake'}
    {'lake'           }
    {'land'           }
```

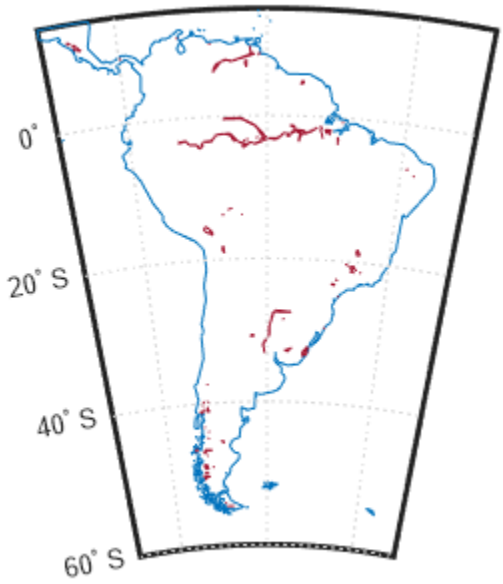
Display Land Areas and Lakes

Create two subtables from the geospatial table. Create the first subtable from the table rows that represent land areas. Create the second subtable from the table rows that represent lakes.

```
idxL1 = (levels == 1);
L1 = GT(idxL1,:);

idxL2 = (levels == 2);
L2 = GT(idxL2,:);
```

Display the land areas and the lakes on a map. Display the land areas using polygons with blue outlines, and display the lakes using polygons with red outlines. Note that the lake polygons include other extended bodies of water.

```
figure
worldmap(latlim,lonlim)
mlabel off
geoshow(L1,EdgeColor="#0072BD",FaceColor="none")
geoshow(L2,EdgeColor="#A2142F",FaceColor="none")
```



Merge Land Areas and Lakes

Merge the land area polygons and the lake polygons, such that the result is a collection of land area polygons with holes that represent lakes.

For each row of the land area and lake geospatial tables:

- Get the bounding boxes for the land area and lake polygons.
- Determine if the bounding boxes overlap by using the `boxesIntersect` on page 2-86 helper function. This step improves the performance of the code by omitting calculations on nonoverlapping polygon shapes.
- If the bounding boxes overlap, create holes in the land area polygon by finding the difference of the land area polygon and the lake polygon.
- In the land area geospatial table, replace the original land area polygon with the updated land area polygon.

```
for landIDX = 1:height(L1)
    % land area bounding box
    landBBOX = L1.BoundingBox{landIDX};
    for lakeIDX = 1:height(L2)
        % lake bounding box
        lakeBBOX = L2.BoundingBox{lakeIDX};
        % determine if bounding boxes overlap
        if boxesIntersect(landBBOX,lakeBBOX)
            landShape = L1.Shape(landIDX);
            lakeShape = L2.Shape(lakeIDX);
            % calculate difference of land area and lake polygons
            mergedShape = subtract(landShape,lakeShape);
            % replace original land area polygon
            L1.Shape(landIDX) = mergedShape;
        end
```

```
    end
end
```

Write Merged Data to Shapefile

Write the updated geospatial table to a shapefile. Note that the shapefile consists of multiple files: a main file (`.shp`), an index file (`.shx`), and an attribute file (`.dbf`).

```
filename = "gshhs_c_SouthAmerica.shp";
shapewrite(L1,filename)
```

Validate Shapefile

Validate the shapefile by reading and displaying the data stored in the file.

Read the shapefile into a new geospatial table.

```
GT2 = readgeotable("gshhs_c_SouthAmerica.shp",CoordinateSystemType="geographic");
```

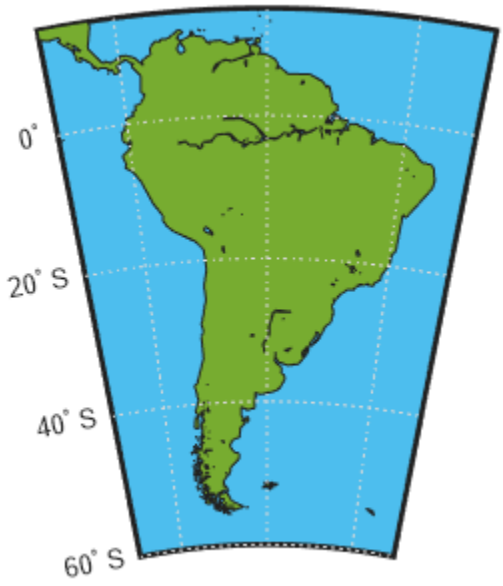
Display the variables names of the new table. Note that, due to character limits in the attribute file (`.dbf`) specification, the original attribute names `FormatVersion` and `CrossesGreenwich` are truncated to 11 characters.

```
GT2.Properties.VariableNames'
```

```
ans = 13×1 cell
    {'Shape'      }
    {'South'      }
    {'North'      }
    {'West'       }
    {'East'       }
    {'Area'       }
    {'Level'      }
    {'LevelString'}
    {'NumPoints'  }
    {'FormatVersi'}
    {'Source'     }
    {'CrossesGree'}
    {'GSHHS_ID'   }
```

Display the merged data on a map. Note that the land area polygons contain holes that represent lakes and other extended bodies of water.

```
figure
worldmap(latlim,lonlim)
ax = gca;
setm(ax,"FFaceColor","#4DBEEE")
mlabel off
geoshow(GT2,FaceColor="#77AC30")
```



References

[1] "GSHHG - A Global Self-Consistent, Hierarchical, High-Resolution Geography Database." Accessed October 22, 2024. https://www.soest.hawaii.edu/pwessel/gshhg/.

Helper Function

The `boxesIntersect` helper function returns a logical `1` if the bounding boxes specified by `bbox1` and `bbox2` intersect, and `0` otherwise. `bbox1` and `bbox2` are 2-by-2 matrices of the form `[lonmin latmin; lonmax latmax]`.

```
function tf = boxesIntersect(bbox1,bbox2)
    tf = ~(any(bbox1(2,:) < bbox2(1,:)) || any(bbox2(2,:) < bbox1(1,:)));
end
```

See Also

Functions

`gshhs` | `struct2geotable` | `subtract` | `shapewrite` | `readgeotable`

Related Examples

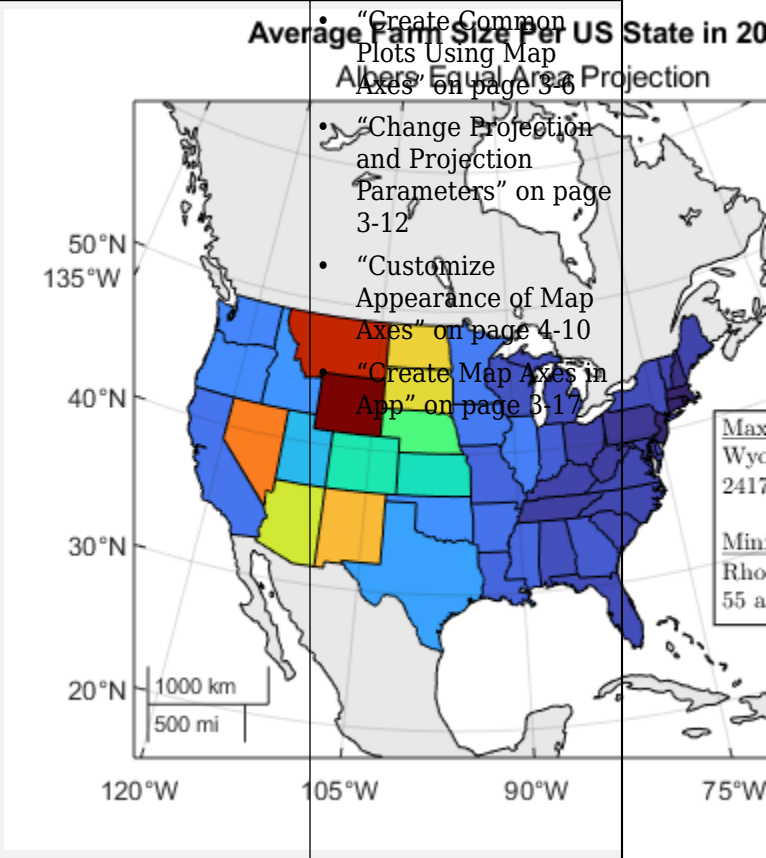
- “Find Geospatial Vector Data” on page 2-8

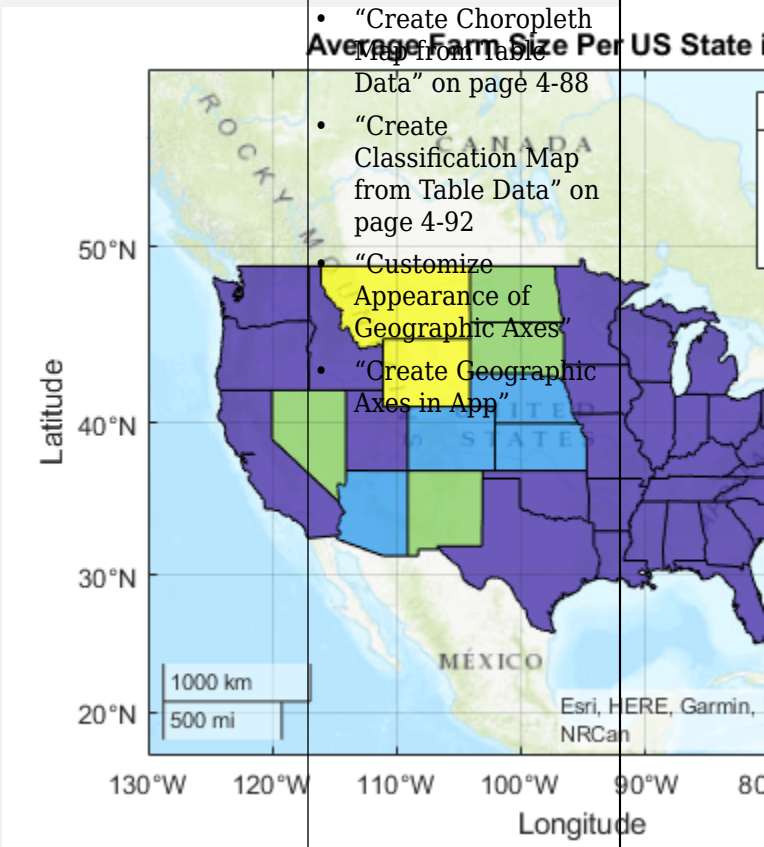
Map Display

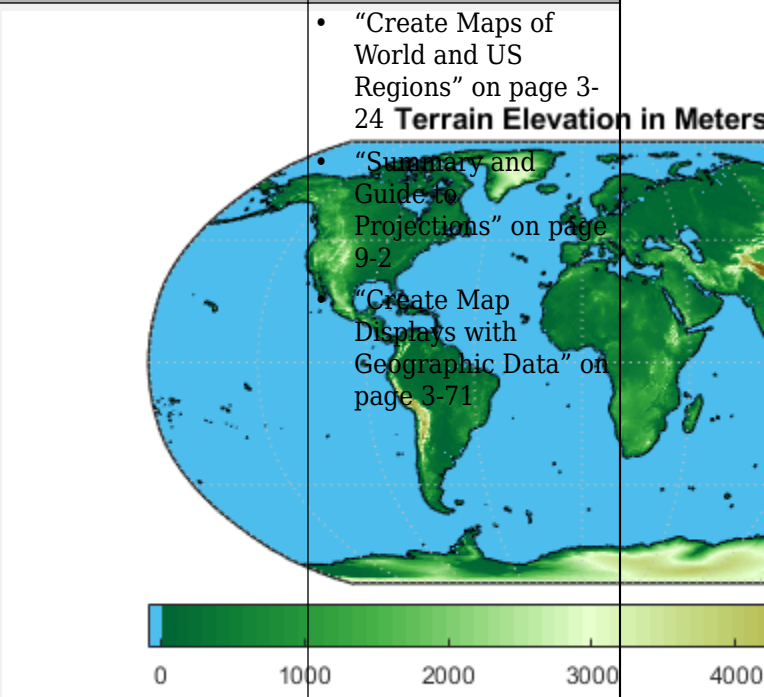
- “Choose a 2-D Map Display” on page 3-2
- “Create Common Plots Using Map Axes” on page 3-6
- “Change Projection and Projection Parameters” on page 3-12
- “Create Map Axes in App” on page 3-17
- “axesm-Based Maps” on page 3-22
- “Create Maps of World and US Regions” on page 3-24
- “Switch Between Projections” on page 3-29
- “Reprojection of Graphics Objects” on page 3-34
- “Change Map Projections When Using geoshow” on page 3-37
- “Display Vector Data as Points and Lines” on page 3-41
- “Display Vector Data as Lines and Patches” on page 3-46
- “Create Maps Using Icons” on page 3-50
- “Control Color and Transparency of Icon Chart” on page 3-56
- “Summary of Polygon Display Functions” on page 3-62
- “Types of Data Grids and Raster Display Functions” on page 3-64
- “Fit Gridded Data to the Graticule” on page 3-65
- “Create 3-D Displays with Raster Data” on page 3-68
- “Create Map Displays with Geographic Data” on page 3-71
- “Create Maps with Data in Projected Coordinate Reference Systems” on page 3-81
- “Interactively Identify Geographic Locations” on page 3-89
- “Interactively Display Text on Maps” on page 3-93
- “Add Data Tips to Point, Line, and Polygon Shapes” on page 3-97
- “Work with Objects by Name” on page 3-107
- “Create an Interactive Map for Selecting Point Features” on page 3-112

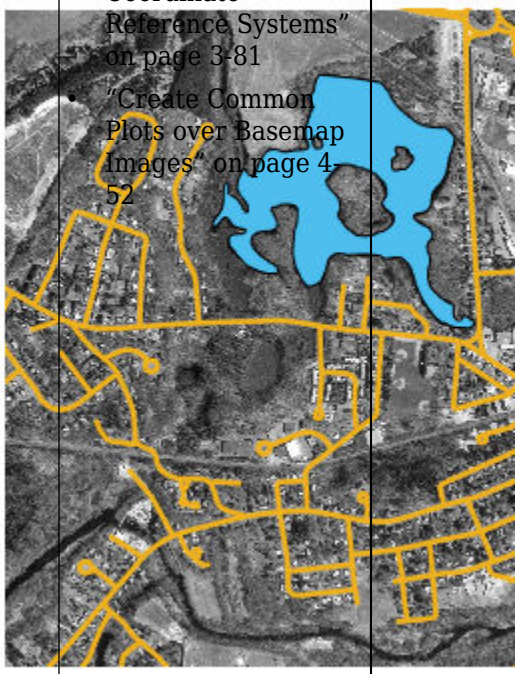
Choose a 2-D Map Display

Mapping Toolbox provides several options for creating 2-D map displays. Choose a map display based on criteria such as the map projection, the type of data, and interactivity options.

Sample Goals	Map Display Option	Sample Map	Related Information
Projection • Use a projected coordinate reference system (CRS) defined by a geospatial data file or by the EPSG. • View a polar region. **Data** • Plot data originating in a mixture of projected and geographic coordinates. • Plot vector data. **Interactivity** • Interactively explore using pan, zoom, and data tips. • Build apps using App Designer.	Use a map axes object. • Create a map axes object from a projected CRS by using the `newmap` function. Specify the projected CRS by using a `projcrs` object. • Plot data into a map axes object by using functions such as `geoplot`, `geoscatter`, `geoiconchart`, and `bubblechart`. • Customize map axes using MATLAB graphics functions. For example, change the geographic limits using `geolimits`, add text using `text`, and change the format of the tick labels using `geotickformat`.		• “Create Common Plots Using Map Axes” on page 3-6 • “Change Projection and Projection Parameters” on page 3-12 • “Customize Appearance of Map Axes” on page 4-10 • “Create Map Axes in App” on page 3-17

Sample Goals	Map Display Option	Sample Map	Related Information
Projection • Use a Web Mercator map projection, which is common for web-based mapping workflows. **Data** • Provide geographic context using high-zoom-level basemaps. • Provide geographic context using data layers imported from Web Map Service (WMS) servers. • Plot data originating in a mixture of projected and geographic coordinates. • Plot vector data. **Interactivity** • Interactively explore using pan, zoom, and data tips. • Build apps using App Designer.	Use a geographic axes object. • Unlike map axes, geographic axes do not require you to create an axes object before plotting data. All geographic axes use a Web Mercator projection. • Plot data into a geographic axes object by using functions such as `geoplot`, `geoscatter`, `geoiconchart`, `geodensityplot`, and `bubblechart`. • Customize geographic axes using MATLAB graphics functions. For example, change the basemap using `geobasemap`, change the geographic limits using `geolimits`, add text using `text`, and change the format of the tick labels using `geotickformat`.		• "Create Choropleth Map from Table Data" on page 4-88 • "Create Classification Map from Table Data" on page 4-92 • "Customize Appearance of Geographic Axes" • "Create Geographic Axes in App"

Sample Goals	Map Display Option	Sample Map	Related Information
Projection • Use a map projection appropriate for a world or US region by specifying the name of the region, for example Europe or California. • Use a predefined projection defined by MATLAB. • View a polar region. **Data** • Plot data in geographic coordinates. • Plot a mixture of vector and raster data.	Use an `axesm`-based map. • Create `axesm`-based maps of world or US regions by using the `worldmap` and `usamap` functions, respectively. Create `axesm`-based maps that use predefined projections by using the `axesm` function. • Plot data into `axesm`-based maps by using functions such as `geoshow` and `contourm`. • Customize `axesm`-based maps by using Mapping Toolbox functions. For example, add text using `textm`, add a scale ruler using `scaleruler`, and add a north arrow using `northarrow`.		• "Create Maps of World and US Regions" on page 3-24 • "Summary and Guide to Projections" on page 9-2 • "Create Map Displays with Geographic Data" on page 3-71

Sample Goals	Map Display Option	Sample Map	Related Information
Projection • Plot data already in a projection using Cartesian axes. **Data** • Plot data in projected coordinates. • Plot a mixture of vector and raster data. • Plot data using a plot type not supported by map axes or geographic axes, for example digraphs and contours. **Interactivity** • Build apps using App Designer.	Use an axes object. • Axes do not require you to create the axes object before plotting data. • Plot map data into an axes object by using functions such as `mapshow`, `digraph`, and `binscatter`. • Customize axes using MATLAB graphics functions. For example, add text using `text` and change the map limits using `xlim` and `ylim`.		• "Create Maps with Data in Projected Coordinate Reference Systems" on page 3-81 • "Create Common Plots over Basemap Images" on page 4-52

See Also

Properties
MapAxes Properties | GeographicAxes Properties | axesm-Based Map Properties | Axes Properties

Create Common Plots Using Map Axes

(Since R2023a)

Map axes objects are a type of axes object, similar to axes objects, geographic axes objects, and polar axes objects. Map axes support several types of plots, including point, line, and polygon plots (`geoplot`), scatter plots (`geoscatter`), icon charts (`geoiconchart`), bubble charts (`bubblechart`), and comet plots (`comet`). You can use map axes with many MATLAB graphics functions, including `gca`, `hold`, `geolimits`, `title`, and `legend`.

Create Map Axes

Unlike other types of axes, map axes require you to create the map axes object before plotting data into it. You can create a map axes object by using the `newmap` or `mapaxes` function.

- Use `newmap` when you want to create a map in the default projection or in a specified projection.
- Use `mapaxes` when you want advanced control of the map, or when you want to include the map in an app.

Create Point, Line, and Polygon Plots

Display points, lines, and polygons on map axes by using the `geoplot` function. You can specify geospatial tables, shape objects, or numeric coordinates as input to the `geoplot` function. When you specify geospatial tables or shape objects, you can plot data in any supported geographic or projected coordinate reference system (CRS).

Set up a new map. By default, map axes use an Equal Earth map projection.

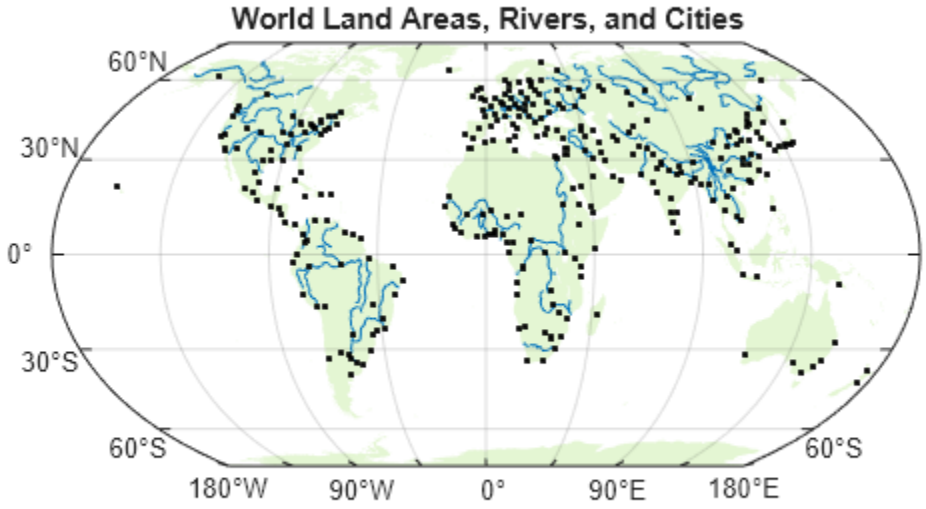
```
figure
newmap
```

Read three shapefiles into the workspace. The files contain world land areas, rivers and cities. Then, display the land areas as green polygons, the rivers as blue lines, and the city locations as black points.

```
land = readgeotable("landareas.shp");
rivers = readgeotable("worldrivers.shp");
cities = readgeotable("worldcities.shp");

geoplot(land,FaceColor=[0.7 0.9 0.5],EdgeColor="none")
hold on
geoplot(rivers,Color=[0 0.4470 0.7410])
geoplot(cities,"k")
```

Add a title.

```
title("World Land Areas, Rivers, and Cities")
```



Create Scatter Plots

Display scatter plots on map axes by using the `geoscatter` function. To use the `geoscatter` function with map axes, you must specify the coordinates using numeric latitudes and longitudes.

Set up a new map using a projected CRS that is appropriate for Europe. For this example, create the CRS using the EPSG code `3035`, which uses a Lambert Azimuthal Equal Area projection method.

```
figure
p = projcrs(3035);
newmap(p)
```

Read a shapefile containing world land areas into the workspace. Provide geographic context for the map by displaying the land areas.

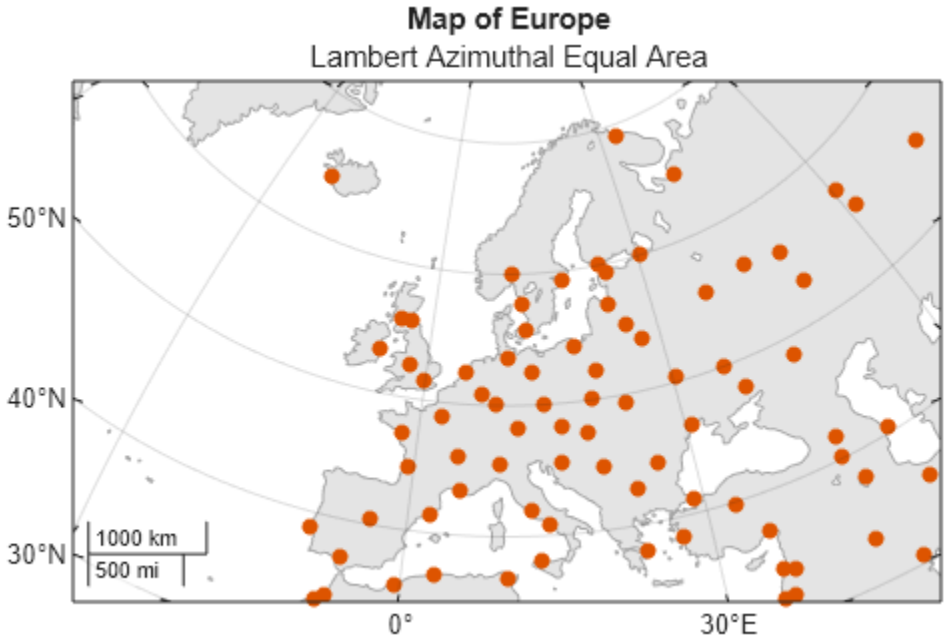
```
land = readgeotable("landareas.shp");
geoplot(land,FaceColor=[0.7 0.7 0.7],EdgeColor=[0.65 0.65 0.65])
hold on
```

Read a shapefile containing the locations of world cities into the workspace. Extract the numeric latitude and longitude coordinates. Then, display the city locations using a scatter plot.

```
cities = readgeotable("worldcities.shp");
lat = cities.Shape.Latitude;
lon = cities.Shape.Longitude;
geoscatter(lat,lon,"filled")
```

Adjust the geographic limits. Then, add a title and subtitle.

```
geolimits([35 72],[-25 45])

title("Map of Europe")
subtitle(p.ProjectionMethod)
```



Create Icon Charts

(Since R2024b)

Display icon charts on map axes by using the `geoiconchart` function. To use the `geoiconchart` function with map axes, you must specify the coordinates using numeric latitudes and longitudes.

Set up a new map using a projected CRS that is appropriate for Massachusetts. For this example, create the CRS using the EPSG code 26986, which uses a Lambert Conic Conformal projection method.

```
figure
p = projcrs(26986);
newmap(p)
```

Create a geospatial table that contains an area of interest (AOI) for Massachusetts. Provide geographic context for the map by displaying the AOI.

```
mass = geocode("Massachusetts");
geoplot(mass,FaceColor=[0.7 0.7 0.7],EdgeColor=[0.65 0.65 0.65])
hold on
```

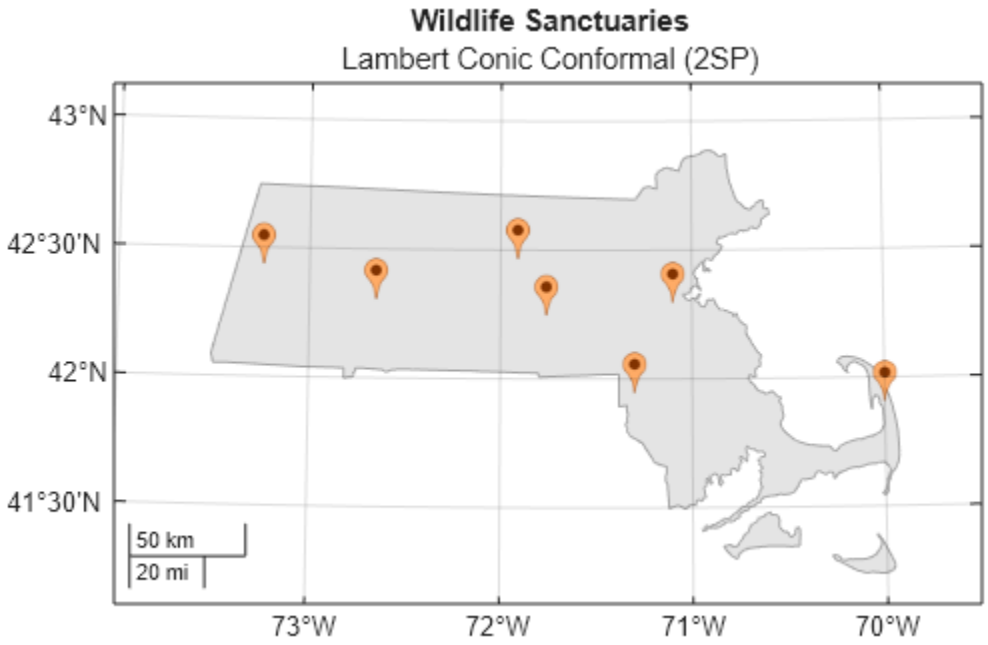
Specify the latitude and longitude coordinates of several wildlife sanctuaries in Massachusetts. Then, display the coordinates using an icon chart. By default, the `geoiconchart` function uses pushpin icons.

```
lat = [42.3 42.24 42.43 42.29 42.46 41.94 41.9];
lon = [-72.65 -71.76 -73.24 -71.1 -71.91 -71.3 -70];
geoiconchart(lat,lon)
```

Adjust the geographic limits. Then, add a title and subtitle.

```
geolimits([41.75 42.5],[-74 -69.5])
```

```
title("Wildlife Sanctuaries")
subtitle(p.ProjectionMethod)
```



Create Bubble Charts

Display bubble charts on map axes by using the `bubblechart` function. To use the `bubblechart` function with map axes, you must specify the coordinates using numeric latitudes and longitudes.

Load Data

Load into the workspace a shapefile containing tsunami events, including attributes such as the maximum heights and the causes. Extract the latitudes, longitudes, maximum heights, and causes. Replace missing causes with `"Unknown Cause"`.

```
tsunamis = readgeotable("tsunamis.shp",CoordinateSystemType="geographic");

lat = tsunamis.Shape.Latitude;
lon = tsunamis.Shape.Longitude;
maxheight = tsunamis.Max_Height;
c = categorical(tsunamis.Cause);
c = fillmissing(c,"constant","Unknown Cause");
```

Set Up Map

This example adds multiple legends to the bubble chart. To manage the alignment of the legends, use a tiled chart layout.

```
figure
t = tiledlayout(1,1);
nexttile
```

Set up a new map using a projected CRS that is appropriate for Southeast Asia. For this example, create a CRS using the ESRI code `102030`, which uses a Lambert Conic Conformal projection method.

```
p = projcrs(102030,Authority="ESRI");
newmap(p)
```

Read a shapefile containing world land areas into the workspace. Provide geographic context for the map by displaying a subset of the land areas. Avoid including the land areas in the legend by setting the `HandleVisibility` property to `"off"`.

```
land = readgeotable("landareas.shp");
subland = land([1:2,5:18,20:end],:);
geoplot(subland,HandleVisibility="off",FaceColor=[0.7 0.7 0.7],EdgeColor=[0.65 0.65 0.65])
hold on
```

Plot Data

Display the data using a bubble chart. To create a legend that illustrates the tsunami causes, use a separate bubble chart for each cause. Specify the bubble sizes using the maximum heights.

```
causes = categories(c);
numCauses = length(causes);

mx = gca;

for k = 1:numCauses
    cause = causes(k);
    idx = c == cause;
    bubblechart(mx,lat(idx),lon(idx),maxheight(idx),DisplayName=string(cause))
end
```

Note that, to use the `bubblechart` function with map axes, you must specify the map axes as the target.

Customize Plot

Customize the plot by changing the bubble sizes, adding legends, changing the geographic limits, and adding a subtitle and title.

Specify the minimum and maximum bubble sizes (in points).

```
bubblesize([3 30])
```

Add two legends. Illustrate the bubble colors using a legend, and illustrate the bubble sizes using a bubble legend. Store each legend object by specifying an output argument for the `bubblelegend` and `legend` functions. Then, move the legends to the bottom outer tile of the tiled chart layout by setting the `Layout.Tile` property on each object to `"south"`.

```
lgd = legend;
title(lgd,"Cause")
blgd = bubblelegend("Maximum Height (m)");

lgd.Layout.Tile = "south";
blgd.Layout.Tile = "south";
```

Change the bubble size limits to reflect the sizes of bubbles in Southeast Asia.

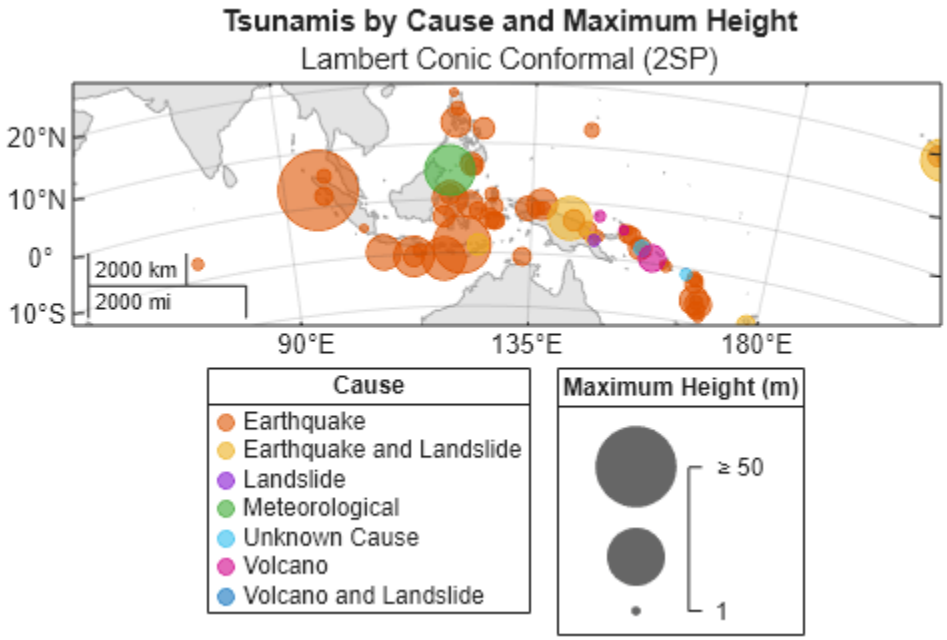
```
bubblelim([1 50])
```

Adjust the geographic limits.

```
geolimits([-20 20],[90 170])
```

Add a title and subtitle.

```
title("Tsunamis by Cause and Maximum Height")
subtitle(p.ProjectionMethod)
```



See Also

Functions
`newmap` | `geoplot` | `geoscatter` | `geoiconchart` | `bubblechart`

Properties
MapAxes Properties

Related Examples

- “Customize Appearance of Map Axes” on page 4-10
- “Change Projection and Projection Parameters” on page 3-12
- “Create Geospatial Tables” on page 6-2

Change Projection and Projection Parameters

Map axes objects use projected coordinate reference systems (CRSs) to transform geographic (latitude-longitude) coordinates to projected (*xy*) coordinates. Projected CRSs consist of a geographic CRS, a projection method, and projection parameters.

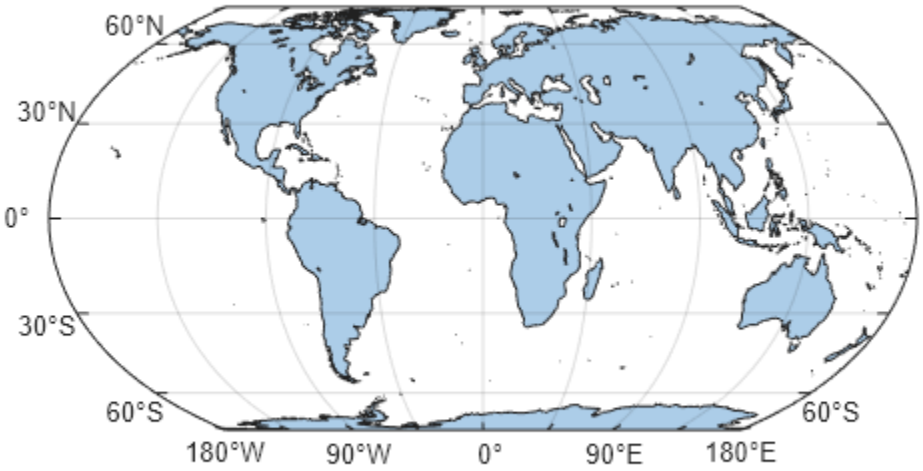
You can change the map projection for a map axes object by changing the projected CRS or by changing the projection parameters. Map axes automatically reproject plotted data when you change the projected CRS or projection parameters.

Change Projected CRS

Change the map projection for a map axes object by changing the projected CRS.

Plot world land areas using the default map axes.

```
figure
newmap

land = readgeotable("landareas.shp");
geoplot(land)
```



View information about the projected CRS stored in the map axes. By default, map axes use the Equal Earth projection method. This projection method preserves area.

```
mx1 = gca;
mx1.ProjectedCRS
```

```
ans =
  projcrs with properties:
```

```
                    Name: "WGS 84 / Equal Earth Greenwich"
            GeographicCRS: [1×1 geocrs]
         ProjectionMethod: "Equal Earth"
               LengthUnit: "meter"
     ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

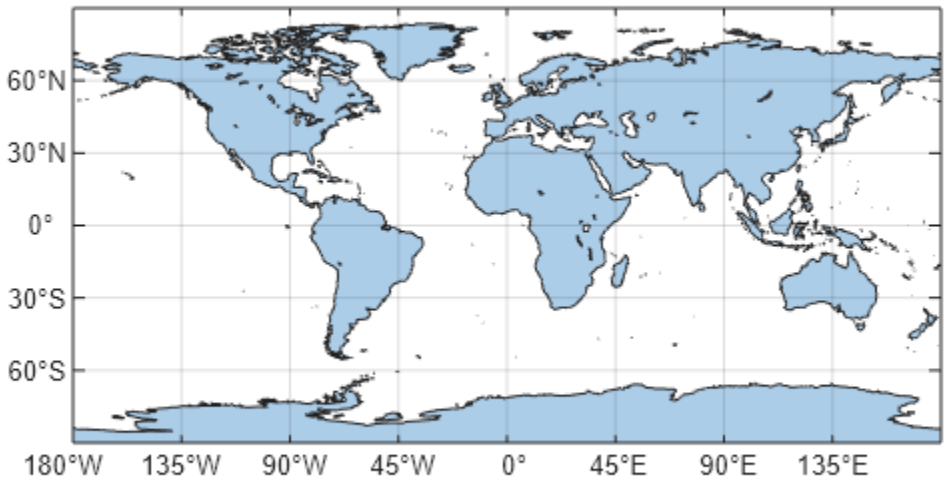
Create a projected CRS object that uses an Equidistant Cylindrical projection method. This projection method preserves distances.

```
p = projcrs(4087)
```

```
p =
  projcrs with properties:

                    Name: "WGS 84 / World Equidistant Cylindrical"
            GeographicCRS: [1×1 geocrs]
         ProjectionMethod: "Equidistant Cylindrical"
               LengthUnit: "meter"
     ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Change the projected CRS of the map axes. When you change the projected CRS, MATLAB® automatically updates the map and reprojects the plotted data.

```
mx1.ProjectedCRS = p;
```



Change Projection Parameters

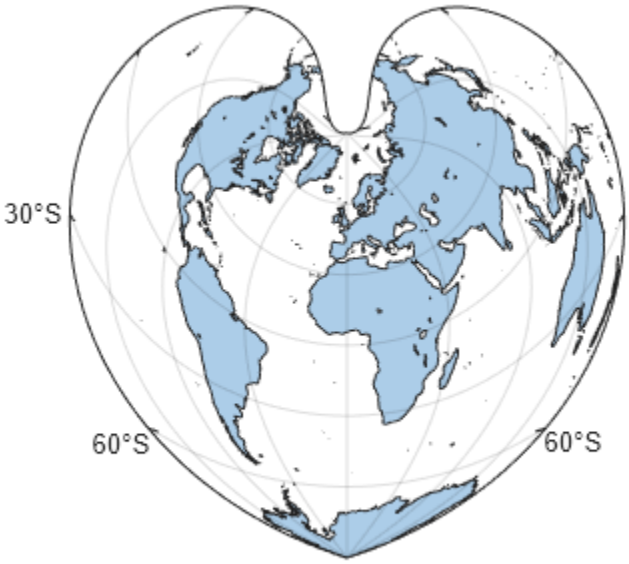
Change the map projection for a map axes object by changing the projected parameters stored in the projected CRS.

Create a projected CRS object that uses the Bonne projection method.

```
p = projcrs(54024,Authority="ESRI")

p =
  projcrs with properties:

                   Name: "World_Bonne"
          GeographicCRS: [1×1 geocrs]
       ProjectionMethod: "Bonne"
             LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Plot the same world land areas in a map axes object that uses the projected CRS.

```
figure
newmap(p)
geoplot(land)
```

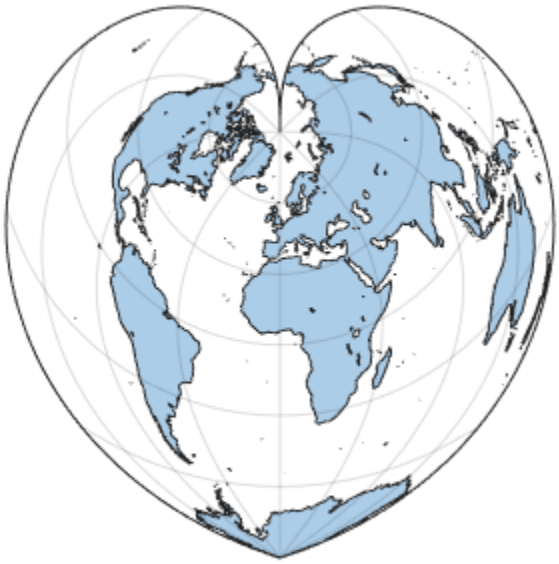


View the projection parameters stored in the projected CRS. For more information about projection parameters, see the “ProjectionParameters” property description on the `projcrs` reference page.

```
mx2 = gca;
mx2.ProjectedCRS.ProjectionParameters

ans =
  ProjectionParameters object with parameters:

     LatitudeOfNaturalOrigin: 60
    LongitudeOfNaturalOrigin: 0
                FalseEasting: 0
               FalseNorthing: 0
```

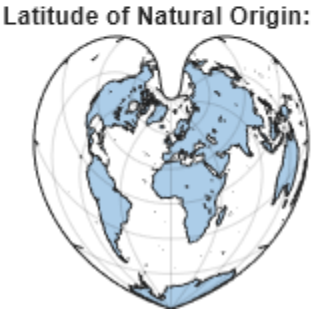
Change the latitude of natural origin to `90`. Valid parameter values depend on the parameter and on the map projection method. When you change a projection parameter, MATLAB automatically updates the map and reprojects the plotted data.

```
mx2.ProjectedCRS.ProjectionParameters.LatitudeOfNaturalOrigin = 90;
```



Create several world maps that use the Bonne projection method in a tiled chart layout. Use a different latitude of natural origin for each map.

```
figure
tiledlayout(2,2,Padding="tight");

for latNaturalOrigin = [1 10 60 90]
    nexttile
    newmap(p)
    geoplot(land)

    mx = gca;
    mx.ProjectedCRS.ProjectionParameters.LatitudeOfNaturalOrigin = latNaturalOrigin;

    mx.FontColor = "none";
    title("Latitude of Natural Origin: " + latNaturalOrigin,ColorMode="auto")
end
```



See Also

Functions
`newmap` | `geoplot`

Objects
`projcrs`

Properties
MapAxes Properties

Related Examples

- “Create Common Plots Using Map Axes” on page 3-6

Create Map Axes in App

This example shows how to include map axes in apps created using App Designer. The example also shows how to include app components that control the map projection and the visibility of plotted data.

To view an app created using the steps in this example, see App That Contains Map Axes on page 3-20.

Create New App

Open App Designer and click **2-Panel App with Auto-Reflow**.

```
appdesigner
```

Create App Components

Add components to the left panel that control the map projection and the visibility of plotted data.

In the top-right corner of the center pane, select **Design View** to interactively add app components. Find the app components by browsing the **Component Library**. Change the appearance and behavior of app components by editing properties in the component tab of the **Component Browser**.

Create a component that controls the map projection.

1. Add a radio button group to the left panel.
2. Change the title of the group to `Map Projection`.
3. Change the text of the radio buttons to `Equal Earth`, `Winkel Tripel`, and `Mercator`.

Create a component that controls the visibility of plotted coastlines, rivers, and cities.

1. Add a panel to the left panel.
2. Change the title of the new panel to `Data`.
3. Add three check boxes to the `Data` panel.
4. Change the text of the check boxes to `Coastlines`, `Rivers`, and `Cities`.
5. Check each check box. For each check box, in the right pane, select **Value**.

Set Up Map

Add the map to the right panel.

In the top-right corner of the center pane, select **Code View** to add the map axes object, load and plot the data, and specify the map projections.

Create a map axes object.

1. Add a private property that stores the map axes object. Storing objects in properties enables you to access the objects throughout your app code. In the **Editor** tab, click **Property** and then **Private Property**. Then, change the property name in the `properties` block to `MapAxesObject`. Enable the app to validate the property value by including the class name `map.graphics.axis.MapAxes`.

```
properties (Access = private)
        MapAxesObject                   map.graphics.axis.MapAxes
 end
```

2 Add this code that creates a map when you start the app. In the **Editor** tab, click **Callback**. Set **App** to the application and **Callback** to `StartupFcn`. Add code to create a map axes in the right panel and store the map axes in the `MapAxesObject` property. Disable panning and zooming by updating the interaction options of the map axes. Then, provide geographic context by displaying world land areas in the map axes.

```
function startupFcn(app)
    % Create map axes
    app.MapAxesObject = mapaxes(app.RightPanel,NextPlot="add",Color="#A6A6A6");

    % Disable panning and zooming
    app.MapAxesObject.InteractionOptions.PanSupported = "off";
    app.MapAxesObject.InteractionOptions.ZoomSupported = "off";

    % Display land areas
    land = readgeotable("landareas.shp");
    geoplot(app.MapAxesObject,land,FaceColor="#d9d9d9",FaceAlpha=1, ...
        EdgeColor="none",HandleVisibility="off",PickableParts="none")
end
```

Plot the coastlines, rivers, and cities data.

1 Add private properties that store the plot objects. In the **Editor** tab, select **Property** and then **Private Property**. Then, change the property name in the `properties` block to `CoastlinesPolygonObject`. Repeat this process to add `RiversLineObject` and `CitiesPointObject` properties.

```
properties (Access = private)
        % Previous code in properties block
        % ...

        CoastlinesPolygonObject         map.graphics.chart.primitive.Polygon
        RiversLineObject                map.graphics.chart.primitive.Line
        CitiesPointObject               map.graphics.chart.primitive.Point
 end
```

2 Add this code that plots data when you start the app. Within the `StartupFcn` callback, add code to plot the coastlines, rivers, and cities data in the map axes. Store the plots in the `CoastlinesPolygonObject`, `RiversLineObject`, and `CitiesPointObject` properties, respectively.

```
function startupFcn(app)
    % Create map axes
    % ...

    % Disable panning and zooming
    % ...

    % Display land areas
    % ...

    % Display coastlines, rivers, and cities
    rivers = readgeotable("worldrivers.shp");
    cities = readgeotable("worldcities.shp");
```

```
    app.CoastlinesPolygonObject = geoplot(app.MapAxesObject,land, ...
        EdgeColor="k",FaceColor="none");
    app.RiversLineObject = geoplot(app.MapAxesObject,rivers,Color="#0072BD");
    app.CitiesPointObject = geoplot(app.MapAxesObject,cities, ...
        MarkerEdgeColor="#A2142F");
end
```

Specify the map projections using projected coordinate reference system (CRS) objects. Create the objects by using the `projcrs` function.

1. Add private properties that store the projected CRS objects. In the **Editor** tab, select **Property** and then **Private Property**. Then, change the property name in the `properties` block to `EqualEarthProjectedCRS`. Repeat this process to add `WinkelTripelProjectedCRS` and `MercatorProjectedCRS` properties.

   ```
   properties (Access = private)
           % Previous code in properties block
           % ...

           EqualEarthProjectedCRS          projcrs
           WinkelTripelProjectedCRS        projcrs
           MercatorProjectedCRS            projcrs
    end
   ```

2. Add this code that creates the projected CRS objects when you start the app. In the `StartupFcn` callback, add code to create `projcrs` objects that use the Equal Earth, Winkel Tripel, and Mercator projections. Store the objects in the `EqualEarthProjectedCRS`, `WinkelTripelProjectedCRS`, and `MercatorProjectedCRS` properties, respectively.

   ```
   function startupFcn(app)
       % Create map axes
       % ...

       % Disable panning and zooming
       % ...

       % Display land areas
       % ...

       % Display coastlines, rivers, and cities
       % ...

       % Create projected CRSs
       app.EqualEarthProjectedCRS = projcrs(8857);
       app.WinkelTripelProjectedCRS = projcrs(54042,Authority="ESRI");
       app.MercatorProjectedCRS = projcrs(54004,Authority="ESRI");
   end
   ```

Program Component Behavior

Use **Code View** to program the behaviors of the radio button group and the check boxes.

1. Add this code that changes the projected CRS when you change the radio button selection. In the **Editor** tab, click **Callback**. Then, set **Component** to `MapProjectionButtonGroup` and **Callback** to `SelectionChangedFcn`. Add code that changes the projected CRS of the map axes depending on the selected radio button.

```
function MapProjectionButtonGroupSelectionChanged(app, event)
    % Change projected CRS of map axes based on selected map projection
    selectedButton = app.MapProjectionButtonGroup.SelectedObject;
    if selectedButton == app.EqualEarthButton % equal earth
        app.MapAxesObject.ProjectedCRS = app.EqualEarthProjectedCRS;
    elseif selectedButton == app.WinkelTripelButton % winkel tripel
        app.MapAxesObject.ProjectedCRS = app.WinkelTripelProjectedCRS;
    elseif selectedButton == app.MercatorButton % mercator
        app.MapAxesObject.ProjectedCRS = app.MercatorProjectedCRS;
    end
end
```

2 Add this code that toggles that visibility of the plots when you check or uncheck the check boxes. In the **Editor** tab, click **Callback**. Then, set **Component** to `CoastlinesCheckBox` and **Callback** to `ValueChangedFcn`. Add code that changes the visibility of the coastlines plot depending on the value of the check box.

```
function CoastlinesCheckBoxValueChanged(app, event)
    % Change visibility of coastlines plot based on check box
    value = app.CoastlinesCheckBox.Value;
    app.CoastlinesPolygonObject.Visible = value;
end
```

Repeat this step to add `ValueChangedFcn` callbacks to the rivers check box, `RiversCheckBox`, and cities check box, `CitiesCheckBox`.

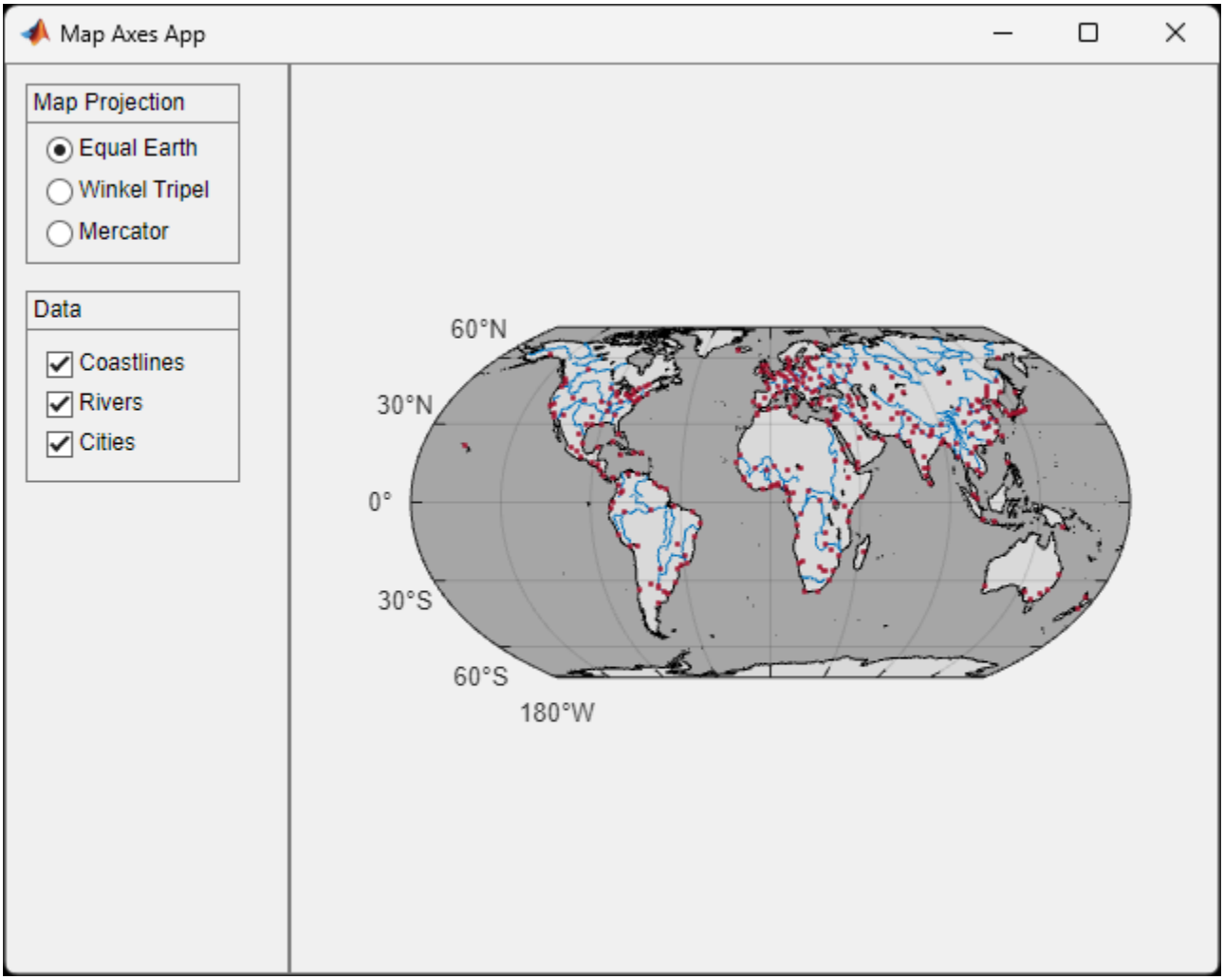
Run the App

To save and run the app, in the **Editor** tab, click **Run**. Change the map projection by clicking the radio buttons. Toggle the visibility of the coastlines, rivers, and cities by clicking the check boxes.

Example: App That Contains Map Axes

This app plots world coastlines, rivers, and cities into a map axes. Radio buttons enable you to change the map projection. Check boxes enable you to toggle the visibility of the plotted data.

To run the app, in the **Editor** tab, click **Run**. Alternatively, enter the name of the app, `MapAxesApp`, in the Command Window.



See Also

Functions
`mapaxes` | `readgeotable` | `geoplot`

Properties
MapAxes Properties | MapAxesInteractionOptions Properties

Related Examples

- “Create and Run a Simple App Using App Designer”
- “App Building Components”
- “Manage Code in App Designer Code View”

axesm-Based Maps

When you create a map, you can use one of the Mapping Toolbox built-in user interfaces (UIs), or you can build the graphic with MATLAB and Mapping Toolbox functions. Many MATLAB graphics are built using the `axes` function:

```
axes
axes('PropertyName',PropertyValue,...)
axes(h)
h = axes(...)
```

Mapping Toolbox functions include an extended version of the `axes` function, called the `axesm` function. Maps created with the `axesm` function, called *`axesm`-based maps* (previously referred to as map axes) share all properties associated with regular axes, and they include information about the current coordinate system (map projection), as well as data to define the map grid and its labeling, the map frame and its limits, scale, and other properties. For complete descriptions of all `axesm`-based map properties, see the axesm-Based Map Properties reference page.

The syntax of `axesm` is similar to that of `axes`:

```
axesm
axesm(PropertyName,PropertyValue,...)
axesm(ProjectionFcn,PropertyName,PropertyValue,...)
```

The `axesm` function without arguments brings up a UI that lists all supported projections and assists in defining their parameters. You can also summon this UI with the `axesmui` function once you have created an `axesm`-based map.

The figure window created using `axesm` contains the same set of tools and menus as any MATLAB figure. By default, the figure window is blank, even if there is map data in your workspace. You can toggle certain properties, such as grids, frames, and axis labels, by right-clicking in the figure window to obtain a pop-up menu.

Most `axesm`-based map functions are similar to MATLAB plotting functions, except they accept data with geographic/geodetic coordinates (latitudes and longitudes) instead of Cartesian and polar coordinates. `axesm`-based map functions typically have the same names as their MATLAB counterparts, with the addition of an `'m'` suffix (for maps). For example, the `axesm`-based map analog to the `plot` function is `plotm`.

`axesm`-based maps manage most of the details in displaying a map. They project your data, cut and trim the data to specified limits, and display the maps at various scales. With the toolbox you can also add customary cartographic elements, such as a frame, grid lines, coordinate labels, and text labels, to your displayed map. If you change your projection properties, or even the projection itself, some map displays are automatically redrawn with the new settings, undoing any cuts or trims if necessary.

Note In its current implementation, the toolbox maintains the map projection and display properties by storing special data in the `UserData` property of the `axesm`-based map. The toolbox also takes over the `UserData` property of objects projected on `axesm`-based maps. Therefore, never attempt to set the `UserData` property of an `axesm`-based map or an object projected onto an `axesm`-based map. Do not apply the MATLAB `get` function to axes `UserData`, depend on the contents of `UserData` in any way, or apply functions that set or get `UserData` to the `axesm`-based map or the objects mapped on an `axesm`-based map. Only use the `getm` and `setm` to obtain and modify the properties of `axesm`-based maps.

Tips to Working with axesm-Based Maps

- You can list all the names, classes, and IDs of map projections supported by `axesm`-based maps with the `maps` function.
- You can place many types of objects in an `axesm`-based map, such as lines, patches, markers, scale rulers, north arrows, grids, and text. You can use the `handlem` function and its associated UI to list these objects. See the `handlem` reference page for a list of the objects that can occupy an `axesm`-based map and how to query for them.
- You can define multiple independent figures containing `axesm`-based maps, but only one can be active at any one time. Use `axes(obj)` to activate an existing `axesm`-based map.
- `axesm`-based maps created by `axesm` contain projection information in a structure. For an example of what these properties are, type

  ```
  h = axesm('MapProjection','mercator')
  ```

  and then use the `getm` function to retrieve all the `axesm`-based map properties:

  ```
  p = getm(h)
  ```

See Also

`axesm` | `axesmui` | `handlem`

Create Maps of World and US Regions

This topic shows how to create maps of world regions and maps of regions in the United States by using the `worldmap` and `usamap` functions. Each function creates an `axesm`-based map and then automatically selects the projection, limits, and other properties based on the region you specify.

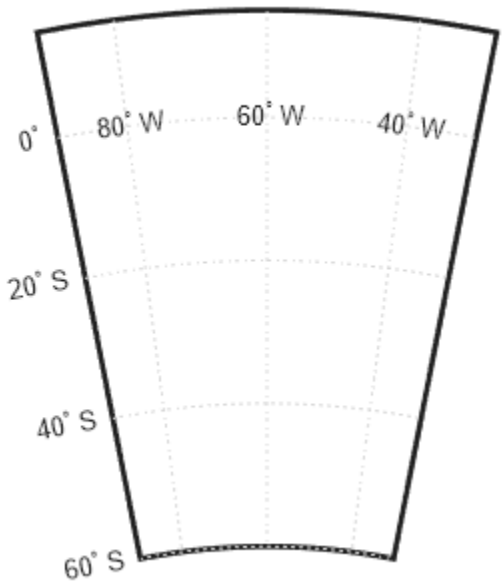
Optionally, after you set up a map using the `worldmap` or `usamap` function, you can customize the map by using the `setm` function.

Create Maps of World Regions

Create a map of a world region by using the `worldmap` function.

Create a map of South America. The `worldmap` function creates an `axesm`-based map and picks an appropriate map projection.

```
figure
worldmap("South America")
```



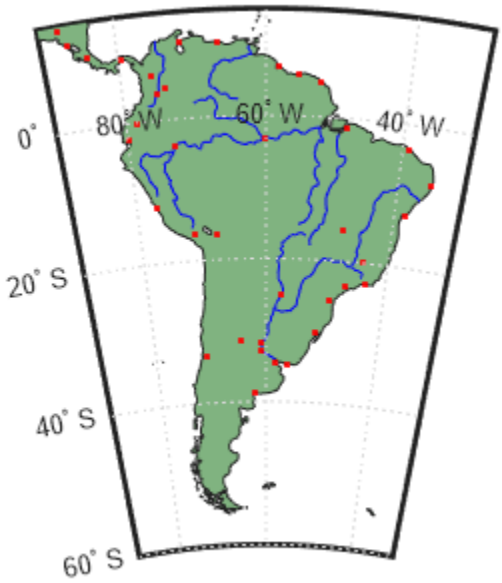
Find the map projection by querying the `MapProjection` property of the `axesm`-based map. The result `'eqdconic'` means that the map uses an Equidistant Conic projection.

```
ax = gca;
getm(ax,"MapProjection")
```

```
ans =
'eqdconic'
```

Display world land areas, rivers, and cities from shapefiles by using the `geoshow` function. Use green for the land areas, blue for the rivers, and black for the cities.

```
geoshow("landareas.shp","FaceColor",[0.5 0.7 0.5])
geoshow("worldrivers.shp","Color","blue")
geoshow("worldcities.shp","Marker",".","Color","red")
```

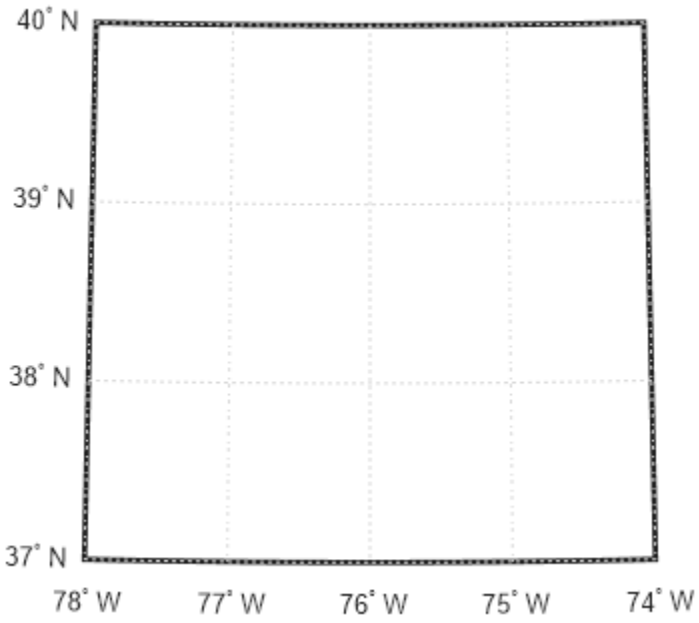


Create Maps of US Regions

Create a map of a region in the United States by using the `usamap` function. The `usamap` function enables you to create maps of the entire United States, the conterminous United States, groups of states, or a single state.

Create a map of the Chesapeake Bay region. The `usamap` function creates an `axesm`-based map and picks an appropriate map projection.

```
figure
usamap([37 40],[-78 -74]);
```



Find the map projection by querying the `MapProjection` property of the `axesm`-based map. The result `'lambert'` means that the map uses a Lambert Conformal Conic projection, which is often used for maps of the conterminous United States.

```
ax = gca;
getm(ax,"MapProjection")
```

```
ans =
'lambert'
```

Read a shapefile of US states into the workspace. The function returns the data in a geospatial table.

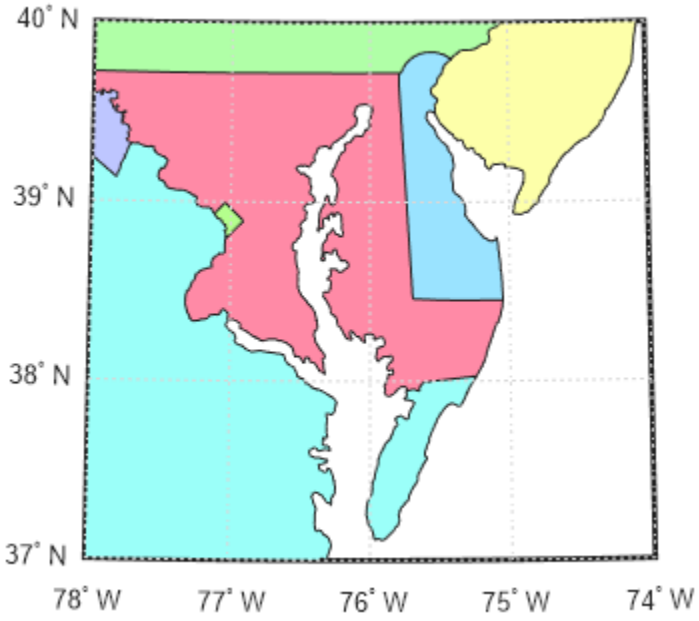
```
states = readgeotable("usastatehi.shp");
```

Define the colors of the states by creating a symbol specification. Create a colormap of random, muted colors by using the `polcmap` function.

```
h = height(states);
faceColors = makesymbolspec('Polygon',{'INDEX',[1 h],'FaceColor',polcmap(h)});
```

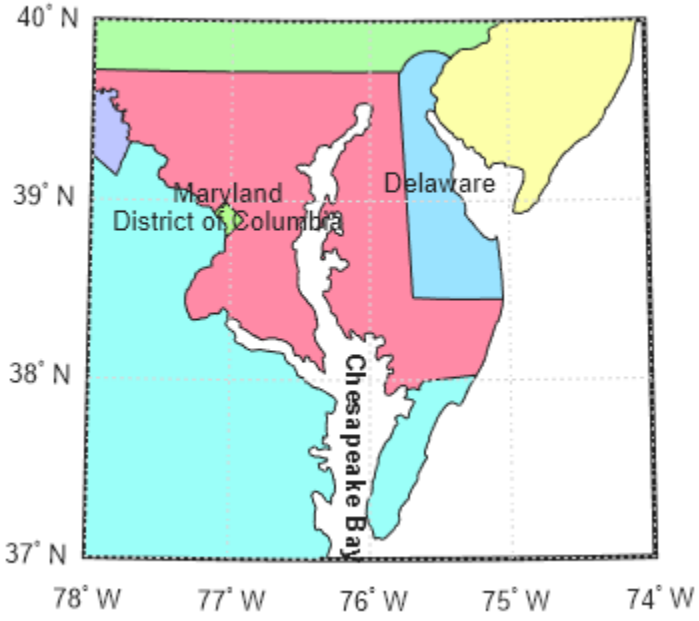
Display the US states by using the `geoshow` function. Specify the color of each state using the symbol specification.

```
geoshow(states,"SymbolSpec",faceColors)
```



Add text labels to the map by using the `textm` function. Get the latitudes, longitudes, and state names from the geospatial table.

```
lat = states.LabelLat;
lon = states.LabelLon;
name = states.Name;
textm(lat,lon,name,"HorizontalAlignment","center")
textm(38.2,-76.1," Chesapeake Bay ","FontWeight","bold","Rotation",270)
```



See Also

Functions
`worldmap` | `usamap` | `setm` | `geoshow`

Properties
axesm-Based Map Properties

Switch Between Projections

Once you create an `axesm`-based map, whether map data is displayed or not, it is possible to change the current projection as well as many of its parameters. Use `setm` to reset the projection. The rest of this section describes the considerations and parameters involved in switching projections in an `axesm`-based map. Additional details are given for doing this with the `geoshow` function in “Change Map Projections When Using geoshow” on page 3-37.

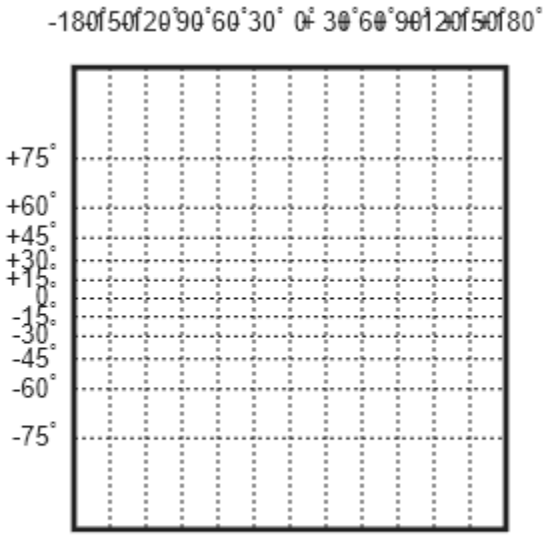
When you switch from one projection to another, `setm` clears out settings that were specific to the earlier projection, updates the map frame and graticule, and generally keeps the map covering the same part of the world—even when switching between azimuthal and non-azimuthal projections. But in some cases, you might need to further adjust the `axesm`-based map properties to achieve proper appearance. Settings that are suitable for one projection might not be appropriate for another. Most often, you'll need to update the positioning of your meridian and parallel labels.

Change Projection Updating Meridian and Parallel Labels

This example shows how to change the projection of an `axesm`-based map and update the meridian and parallel labels.

Create a Mercator projection with meridian and parallel labels.

```
axesm mercator
framem on; gridm on; mlabel on; plabel on
setm(gca,'LabelFormat','signed')
axis off
```



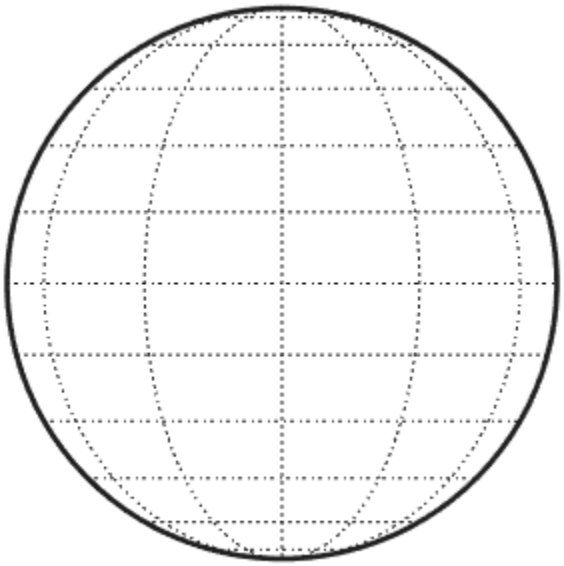
Get the default map and frame latitude limits for the Mercator projection. Note that both the frame and map latitude limits are set to 86 degrees north and south for the Mercator projection to maintain a safe distance from the singularity at the poles.

```
[getm(gca,'MapLatLimit'); getm(gca,'FLatLimit')]
```

```
ans = 2×2

   -86    86
   -86    86
```

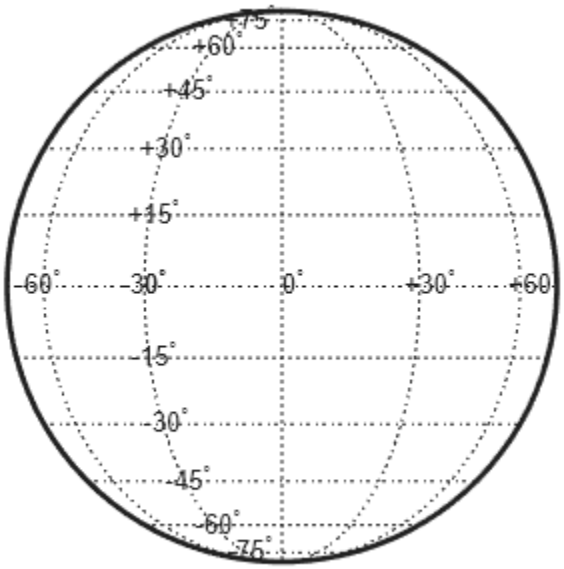
Switch the projection to an orthographic azimuthal.

```
setm(gca,'MapProjection','ortho')
```



Specify new locations for the meridian and parallel labels.

```
setm(gca,'MLabelParallel',0,'PLabelMeridian',-90, ...
   'PLabelMeridian',-30)
```

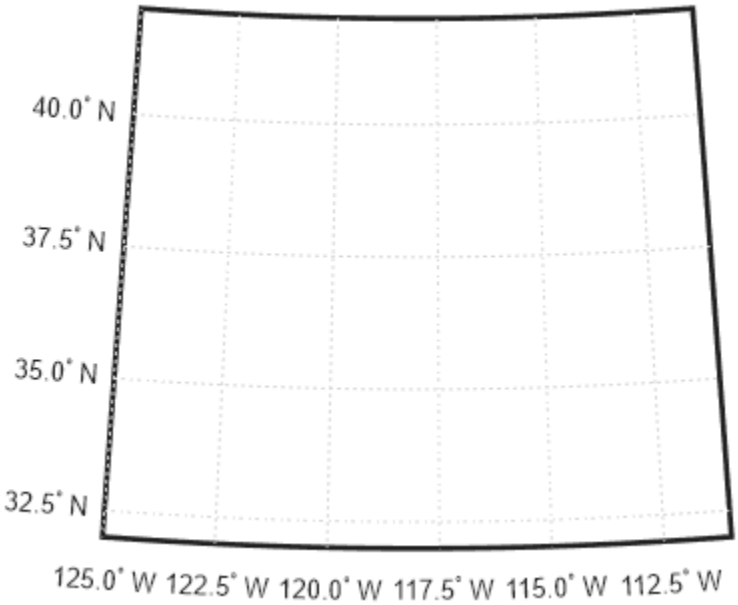


Change Projection Resetting Frame Limits

This example shows how to switch from one projection to another in an `axesm`-based map and reset the origin and frame limits, especially when mapping a small portion of the Earth.

Create an `axesm`-based map for a region of the United States in the Lambert Conformal Conic projection (the default projection for the `usamap` function).

```
latlim = [32 42];
lonlim = [-125 -111];
h = usamap(latlim, lonlim);
```



Read the `usastatehi` shapefile and return a subset of the shapefile contents, as defined by the latitude and longitude limits. The `shaperead` function returns the data in a structure called `states`.

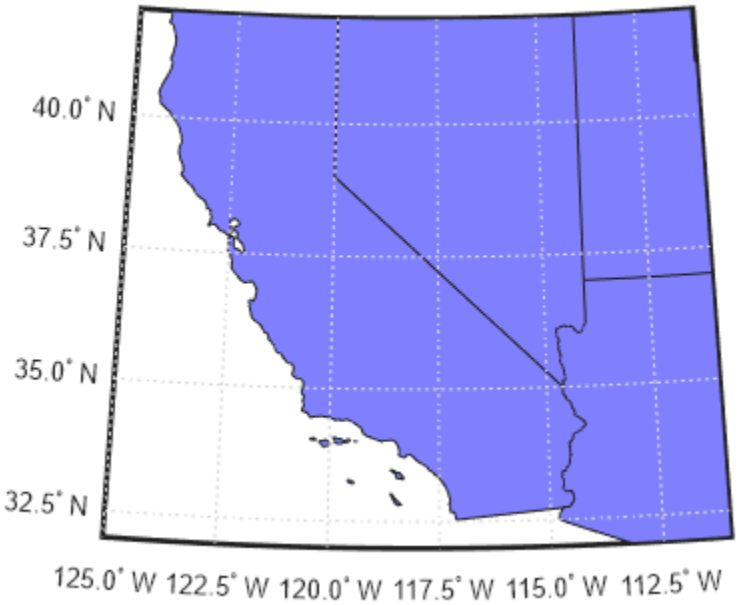
```
states = shaperead('usastatehi.shp', 'UseGeoCoords', true, ...
   'BoundingBox', [lonlim', latlim']);
```

Save the latitude and longitude data from the structure in the vectors `lat` and `lon`.

```
lat = [states.Lat];
lon = [states.Lon];
```

Project patch objects on the map.

```
patchm(lat, lon, [0.5 0.5 1])
```



Change the projection to Lambert Equal Area Azimuthal and reset the origin and frame limits.

```
setm(gca,'MapProjection','eqaazim','Origin',[37 -118], ...
   'FLatLimit',[-Inf 6])
setm(gca,'mlinelocation',2,'plinelocation',2)
tightmap
```

Reprojection of Graphics Objects

Many Mapping Toolbox cartographic functions project features on an `axesm`-based map based on their designated latitude-longitude positions. The latitudes and longitudes are mathematically transformed to *x* and *y* positions using the formulas for the current map projection. If the map projection or its parameters change, objects on an `axesm`-based map can be automatically reprojected to update the map display accordingly.

The table summarizes the four common use cases for changing a map projection in an `axesm`-based map with `setm` or for reprojecting map data plotted on a regular MATLAB axes.

Mapping Use Case	Type of Axes	Reprojection Behavior
Plot geographic (*latitude-longitude*) vector coordinate data or data grid using a Mapping Toolbox function from releases prior to Version 2 (e.g., `plotm`)	`axesm`-based map	Automatic reprojection
Plot geographic vector data with `geoshow`	`axesm`-based map	No automatic reprojection; delete graphics objects prior to changing the projection and redraw them afterwards.
Plot data grids, images, and contours with geographic coordinates with `geoshow`	`axesm`-based map	Automatic reprojection; this behavior could change in a future release
Plot projected (*x*-*y*) vector or raster map data with `mapshow` or with a MATLAB graphics function (e.g., `line`, `contour`, or `surf`)	Regular axes	Manual reprojection (reproject coordinates with `projinv`/`projfwd`); delete graphics objects prior to changing the projection and redraw them afterwards.

You can use `handlem` to help identify which objects to delete when manual deletion is necessary. See “Work with Objects by Name” on page 3-107 for an example of its use.

Auto-Reprojection of Mapped Objects and Its Limitations

Using the `setm` function, you can change the current map projection on the fly if the map display was created in a way that permits reprojection. Note that map displays can contain objects that cannot be reprojected, and may need to be explicitly deleted and redrawn. Automatic reprojection will take place when you use `setm` to modify the `MapProjection` property, or any other `axesm`-based map property from the following list:

- `AngleUnits`
- `Aspect`
- `FalseEasting`
- `FalseNorthing`
- `FLatLimit`
- `FLonLimit`
- `Geoid`
- `MapLatLimit`

- `MapLonLimit`
- `MapParallels`
- `Origin`
- `ScaleFactor`
- `TrimLat`
- `TrimLon`
- `Zone`

Auto-reprojection takes place for objects created with any of the following Mapping Toolbox functions:

- `contourm`
- `contour3m`
- `fillm`
- `fill3m`
- `gridm`
- `linem`
- `meshm`
- `patchm`
- `plotm`
- `plot3m`
- `surfm`
- `surfacem`
- `textm`

The above Mapping Toolbox functions are analogous to standard MATLAB graphics functions having the same name, less the trailing `m`. You can use both types of functions to plot data on an `axesm`-based map, as long as you are aware that the standard MATLAB graphics functions do not apply map projection transformations, and therefore require you to specify positions in map *x*-*y* space.

In general, objects created with `geoshow` or with a combination of calls to `projfwd` followed by ordinary MATLAB graphics functions, such as `line`, `patch`, or `surface`, are *not* automatically reprojected. You should delete such objects whenever you change one or more of the `axesm`-based map properties listed above, and then redisplay them.

If you have preprojected vector or raster map data or read such data from files, you can display it with `mapshow` or standard MATLAB graphics functions, such as `plot` or `mesh`. If its projection is known and is included in the Mapping Toolbox projection libraries, you can use its parameters to project geodata in geographic coordinates to display it in the same axes.

Reprojectability of Maps Generated Using geoshow

If you want to be able to change the projection of a map on the fly, you should not use `geoshow`. Some display functions, such as `patchm` , `fillm`, `displaym`, and `linem`, enable you to reproject vector map data, but `geoshow` does not. That is, when you change an `axesm`-based map projection, with `setm` for example, vector map symbology that was created with `geoshow` will not be

transformed. Gridded data rendered with `geoshow` (when `DisplayType` is `surface`, `texturemap`, or `contour`), however, can be reprojected.

For examples of reprojection behavior with vector data and raster data, see “Change Map Projections When Using geoshow” on page 3-37.

Change Map Projections When Using geoshow

You can display latitude-longitude raster and vector data on `axesm`-based maps by using the `geoshow` function. When you change the projection of an `axesm`-based map that contains raster data plotted by the `geoshow` function, MATLAB reprojects the raster data. However, if the map contains vector data plotted by the `geoshow` function, MATLAB does not reproject the vector data. To reproject vector data, you must remove the vector data from the map and then plot it again.

Change Map Projections When Using geoshow

Change the map projection of an `axesm`-based map after displaying raster and vector data using the `geoshow` function.

Reproject Raster Data

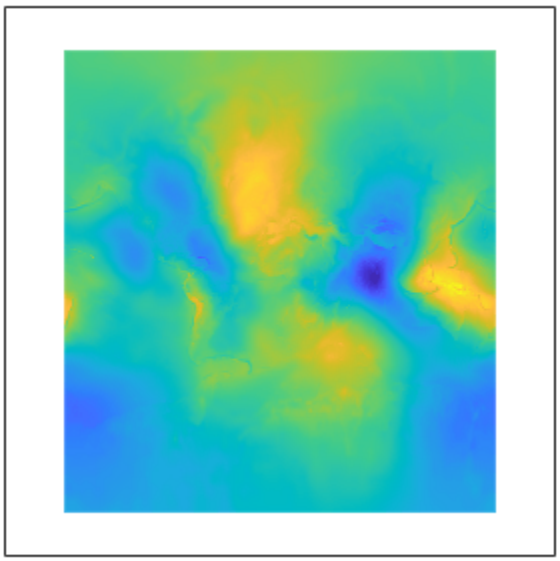
When you change the projection of an `axesm`-based map that contains raster data plotted by the `geoshow` function, MATLAB reprojects the raster data.

Set up an `axesm`-based map that uses a Mercator projection.

```
figure
axesm mercator
```

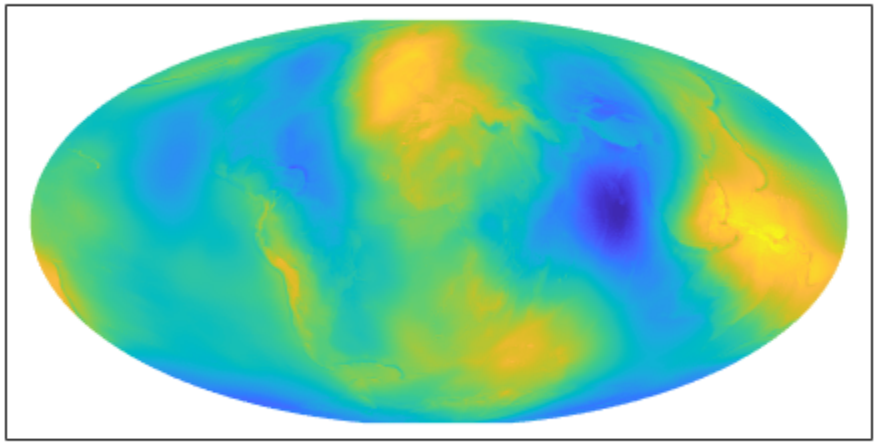
Display raster data on the map by using the `geoshow` function. For this example, display geoid heights from the EGM96 geoid model.

```
[N,R] = egm96geoid;
geoshow(N,R,"DisplayType","surface")
```



Change the map projection to use a Mollweide projection.

```
ax = gca;
setm(ax,"MapProjection","mollweid")
```



Reproject Vector Data

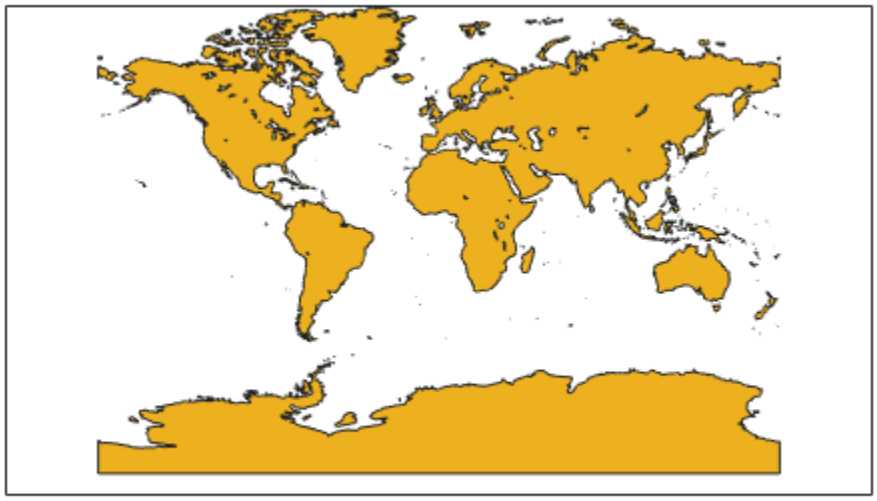
When you change the projection of an `axesm`-based map that contains vector data plotted by the `geoshow` function, MATLAB does not reproject the vector data. To reproject the vector data, you must remove the vector data from the map and then plot it again.

Set up an `axesm`-based map that uses a Miller projection.

```
figure
axesm miller
```

Display vector data on the map by using the `geoshow` function. For this example, display a shapefile containing world land areas. Return the plot objects in `v`.

```
v = geoshow("landareas.shp");
```



To reproject the map, first delete the plot objects. Change the map projection to use an orthographic projection. Then, plot the data again.

```
delete(v)

ax = gca;
setm(ax,"MapProjection","ortho")

geoshow("landareas.shp")
```

See Also

Functions
`axesm`

Properties
axesm-Based Map Properties

Related Examples

- “Reprojection of Graphics Objects” on page 3-34
- “Create Maps of World and US Regions” on page 3-24
- “Summary and Guide to Projections” on page 9-2

Display Vector Data as Points and Lines

Display vector data on maps as points and lines.

This page shows how to create similar maps using map axes (*since R2023a*) and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Prepare Data

Prepare data to use in the examples.

Load a MAT-file containing the coordinates of global coastlines into the workspace. The variables within the MAT-file, `coastlat` and `coastlon`, specify numeric latitude and longitude coordinates. Remove the South Pole by clipping the line data.

```
load coastlines
coast = geolineshape(coastlat,coastlon);
coast = geoclip(coast,[-89 90],[-180 180]);
```

Specify the locations of three cities (Cairo, Rio de Janeiro, and Perth).

```
citylat = [30 -23 -32];
citylon = [32 -43 116];
```

Find the coordinates of two tracks:

- A great circle track from Cairo to Rio de Janeiro
- A rhumb line track from Cairo to Perth

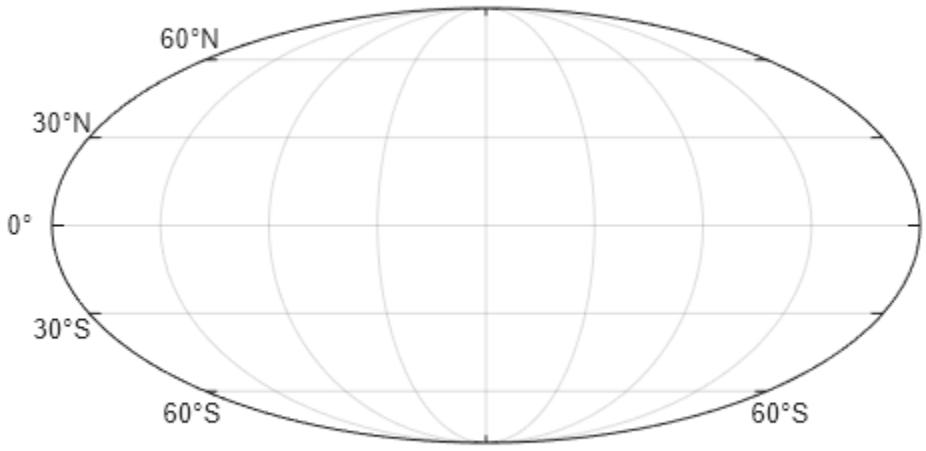
```
[gclat,gclon] = track2("gc",citylat(1),citylon(1),citylat(2),citylon(2));
[rhlat,rhlon] = track2("rh",citylat(1),citylon(1),citylat(3),citylon(3));
```

Create Map Using Map Axes

Display vector data on a map axes object as points and lines.

Set up a map that uses a Mollweide projection. Create the projected coordinate reference system (CRS) using the ESRI code `54009`.

```
figure
proj = projcrs(54009,Authority="ESRI");
mx = newmap(proj);
hold on
```



Display the global coastlines by using the `geoplot` function. Specify the line color as gray.

```
geoplot(coast,Color=[0.4 0.4 0.4])
```

Display the great circle track from Cairo to Rio de Janeiro using a thick line.

```
geoplot(gclat,gclon,LineWidth=2)
```

Display the rhumb line track from Cairo to Perth using a thick, dashed line.

```
geoplot(rhlat,rhlon,"--",LineWidth=2)
```

Display the locations of the cities by using the `geoscatter` function.

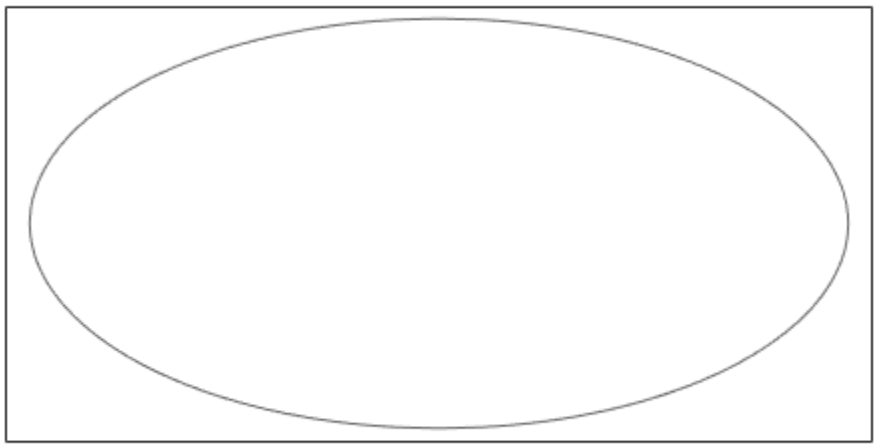
```
geoscatter(citylat,citylon,"filled")
```

Create Map Using axesm-Based Map

Display vector data on an `axesm`-based map as points and lines.

Set up a map that uses a Mollweide projection. Change the color of the frame to gray. Make the line width of the frame thinner.

```
figure
axesm mollweid
framem("FEdgeColor",[0.4 0.4 0.4],"FLineWidth",0.5)
```



Display the global coastlines by using the `plotm` function. Specify the line color as gray.

```
plotm(coastlat,coastlon,"Color",[0.4 0.4 0.4])
```

Display the great circle track from Cairo to Rio de Janeiro using a thick line.

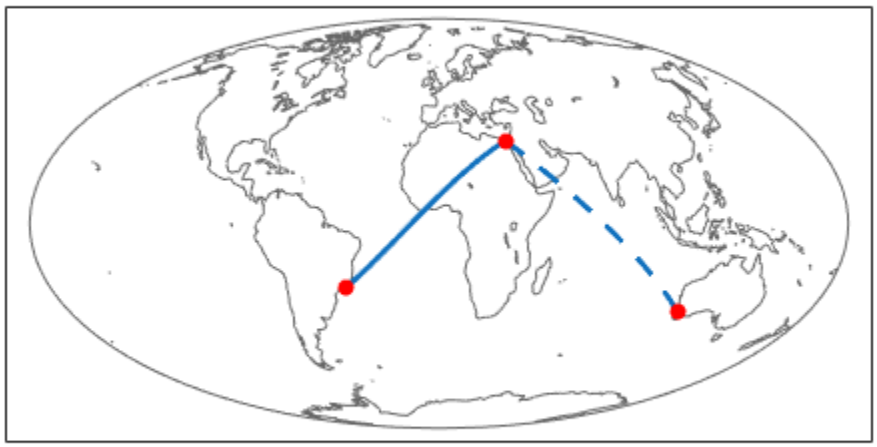
```
plotm(gclat,gclon,"LineWidth",2)
```

Display the rhumb line track from Cairo to Perth using a thick, dashed line.

```
plotm(rhlat,rhlon,"--","LineWidth",2)
```

Display the locations of the cities by using the `scatterm` function.

```
scatterm(citylat,citylon,"r","filled")
```



Alternatively, you can plot geographic data over `axesm`-based maps by using the `geoshow` function.

Tips

To plot data that is already in projected coordinates, use a regular axes object and the `mapshow` function.

See Also

Functions

`newmap` | `axesm`

Properties

MapAxes Properties | axesm-Based Map Properties

Related Examples

- “Display Vector Data as Lines and Patches” on page 3-46

Display Vector Data as Lines and Patches

Display vector data on maps as lines and patches (filled polygons).

This page shows how to create similar maps using map axes (*since R2023a*) and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Prepare Data

Prepare the data to use in the examples.

Load a MAT file containing the latitude and longitude coordinates of several polygons and lines. The file contains the following variables. Note that the data separates individual polygons and lines using `NaN` values.

- `uslat` and `uslon` — Polygons representing the conterminous United States.
- `gtlakelat` and `gtlakelon` — Polygons representing the Great Lakes.
- `statelat` and `statelon` — Lines representing the borders between states.

```
load conus.mat
```

Use the data to create polygon and line shapes.

```
us = geopolyshape(uslat,uslon);
gtlake = geopolyshape(gtlakelat,gtlakelon);
states = geolineshape(statelat,statelon);
```

Read a shapefile containing world rivers into a geospatial table. The table represents the rivers using line shapes in geographic coordinates.

```
riversGT = readgeotable("worldrivers.shp");
```

Clip the line shapes in the geospatial table to the polygon shape that represents the conterminous United States.

```
clipped = geoclip(riversGT.Shape,us);
riversGT.Shape = clipped;
```

Create Map Using Map Axes

Display vector data on a map axes object using lines and polygons.

Set up a map using a projected coordinate reference system (CRS) that is appropriate for the conterminous United States. Create the projected CRS using the ESRI code `102003`.

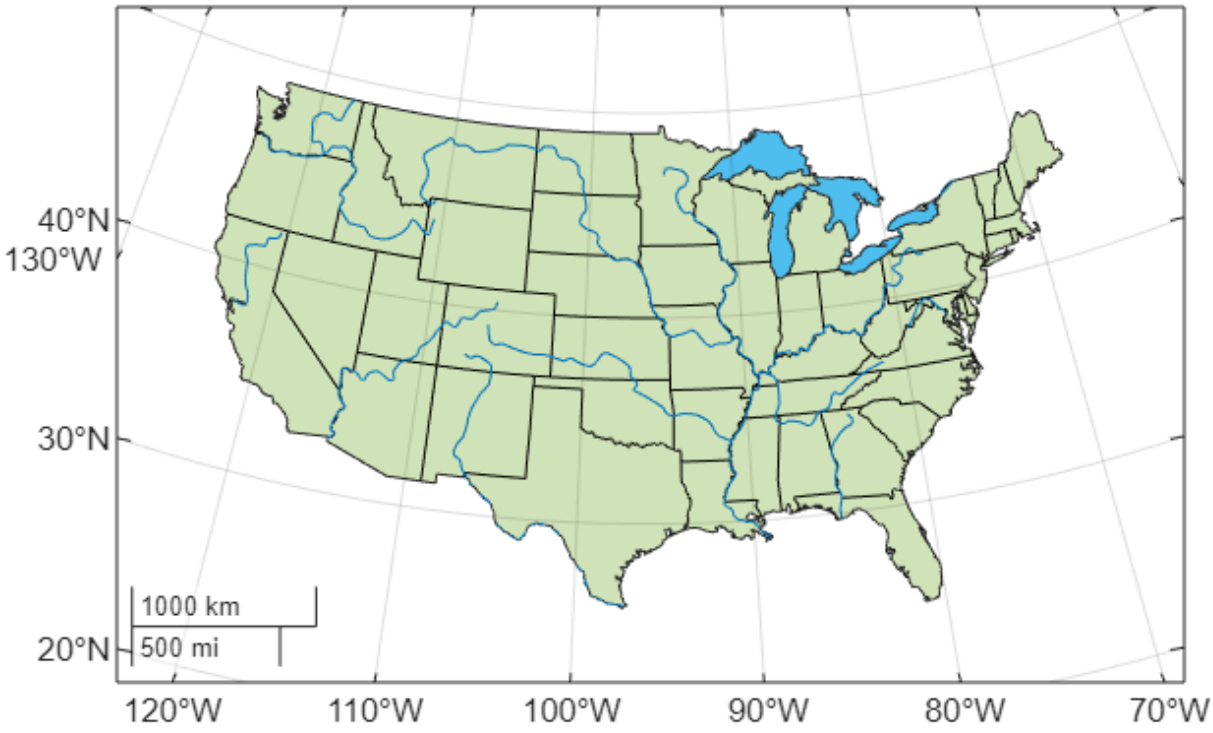
```
figure
proj = projcrs(102003,Authority="ESRI");
newmap(proj)
hold on
```

Display the data on the map by using the `geoplot` function.

- Display the conterminous United States using green polygons.

- Display the Great Lakes using opaque, light blue polygons.
- Display the state boundaries using black lines.
- Display the rivers using blue lines.

```
geoplot(us,FaceColor=[0.4660 0.6740 0.1880])
geoplot(gtlake,FaceAlpha=1,FaceColor=[0.3010 0.7450 0.9330])
geoplot(states,Color="k")
geoplot(riversGT,Color=[0 0.4470 0.7410])
```



Create Map Using `axesm`-Based Map

Display vector data on an `axesm`-based map using lines and patches.

Create a polygon shape that extends outside of the conterminous United States by using the `buffer` function. Then, calculate the latitude and longitude limits to use for the `axesm`-based map by using the `bounds` function.

```
buffered = buffer(us,2);
[latlim,lonlim] = bounds(buffered);
```

Create an `axesm`-based map by using the `axesm` function, an Equal Area Conic projection, and the calculated limits. When you specify map limits, the map automatically centers the projection on an appropriate longitude. Use the default latitudes for the standard parallels by specifying the `MapParallels` argument as an empty matrix.

```
figure
axesm(MapProjection="eqaconic",MapParallels=[], ...
```

```
    MapLatLimit=latlim,MapLonLimit=lonlim)
axis off
```

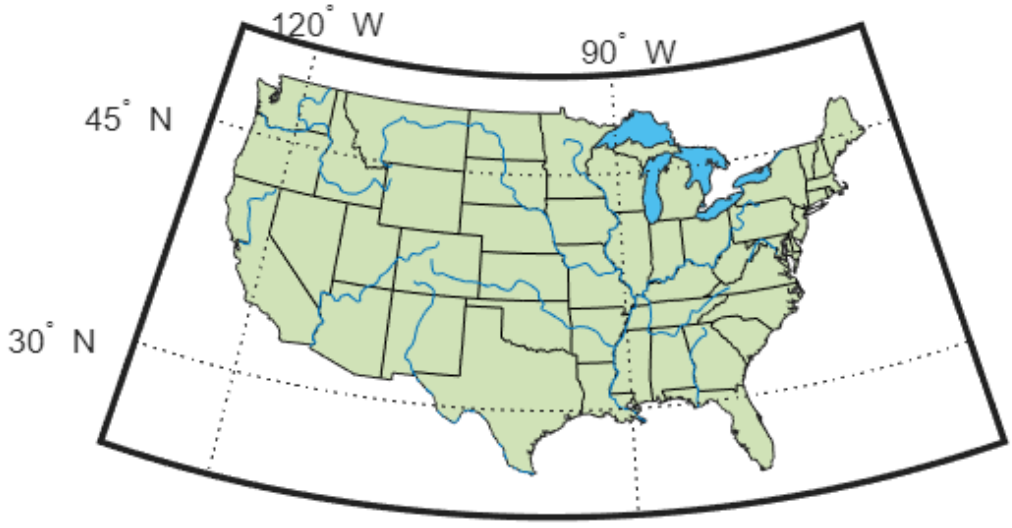
Display the map frame, map grid, meridian labels, and parallel labels.

```
framem
gridm
mlabel
plabel
```

Collect the shape objects into a geospatial table. Then, display the data on the map by using the `geoshow` function.

- Display the conterminous United States using green patches.
- Display the Great Lakes using light blue patches.
- Display the state boundaries using black lines.
- Display the rivers using blue lines.

```
conusGT = table([us; gtlake; states],VariableNames="Shape");

geoshow(conusGT(1,:),FaceColor=[0.8157 0.8863 0.7176])
geoshow(conusGT(2,:),FaceColor=[0.3010 0.7450 0.9330])
geoshow(conusGT(3,:),Color="k")
geoshow(riversGT,Color=[0 0.4470 0.7410])
```



Tips

- Unlike the `axesm` function, the `worldmap` and `usamap` functions automatically select appropriate limits based on the input data.
- To plot data that is already in projected coordinates, use a regular axes object and the `mapshow` function.

See Also

Functions

`axesm` | `geoclip` | `geoshow` | `mapshow` | `geoplot`

Properties

axesm-Based Map Properties

More About

- “Create and Display Polygons” on page 6-17
- “Display Vector Data as Points and Lines” on page 3-41
- “Create Map of Quadrangle Using Cartographic Map Layout” on page 4-16

Create Maps Using Icons

Icon charts enable you to display symbols on maps. Create icon charts in geographic axes or map axes by using the `geoiconchart` function. By default, the function uses a pushpin icon. You can also use custom icons by specifying the name of an image file.

This topic shows several ways to display icons on maps. For example, you can use icons to:

- Represent point data.
- Display quantitative data at representative points.
- Animate movement along a path.
- Annotate a basemap.

Represent Point Data

Use icons to represent the latitude and longitude coordinates of point data.

Read a shapefile containing tsunami events into the workspace as a geospatial table. The table represents the tsunami events using point shapes in geographic coordinates.

```
tsunamis = readgeotable("tsunamis.shp",CoordinateSystemType="geographic");
```

Create a subtable of tsunami events that were caused by volcanoes or by volcanoes and landslides.

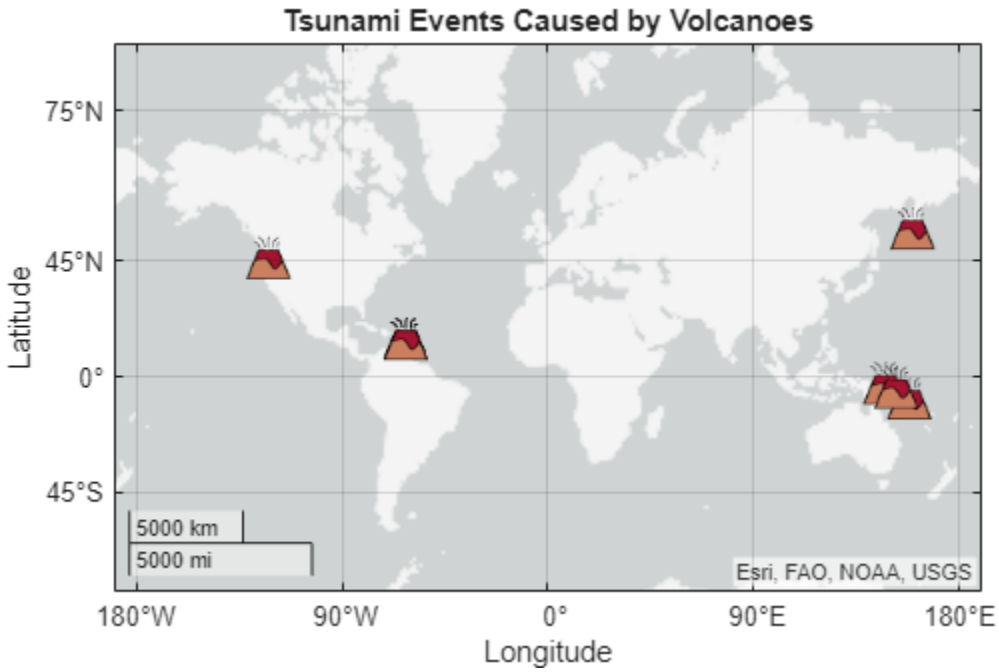
```
idx = contains(tsunamis.Cause,"Volcano");
volcanos = tsunamis(idx,:);
```

Extract the latitude and longitude coordinates from the subtable. Then, display a volcano icon at each point by using the `geoiconchart` function. Return the `IconChart` object in `ic`.

```
lat = volcanos.Shape.Latitude;
lon = volcanos.Shape.Longitude;
ic = geoiconchart(lat,lon,"volcano.png");
```

Zoom out by changing the geographic limits of the map. Add a title.

```
geolimits([-68 81],[-180 180])
title("Tsunami Events Caused by Volcanoes")
```



Display Quantitative Data at Representative Points

Use icons to display quantitative data at representative points. For this example, use pie chart icons to display the agricultural land percentage for several US states.

Import a shapefile of US states into the workspace as a geospatial table. Each table row includes the name of a state, a polygon shape for the state, and the latitude and longitude coordinates of a representative point in the state.

```
states = readgeotable("usastatehi.shp");
```

Create a subtable of nine midwest states: Kansas, Nebraska, South Dakota, North Dakota, Iowa, Missouri, Minnesota, Wisconsin, and Illinois.

```
names = ["Kansas","Nebraska","South Dakota","North Dakota","Iowa","Missouri", ...
    "Minnesota","Wisconsin","Illinois"];
midwest = geocode(names,states);
```

Create a map by displaying the states in a map axes. Specify the projected coordinate reference system (CRS) for the map using the ESRI code 102008, which uses an Albers Equal Area projection.

```
figure
pcrs = projcrs(102008,Authority="ESRI");
newmap(pcrs)
geoplot(midwest,FaceColor=[0.5 0.5 0.5])
```

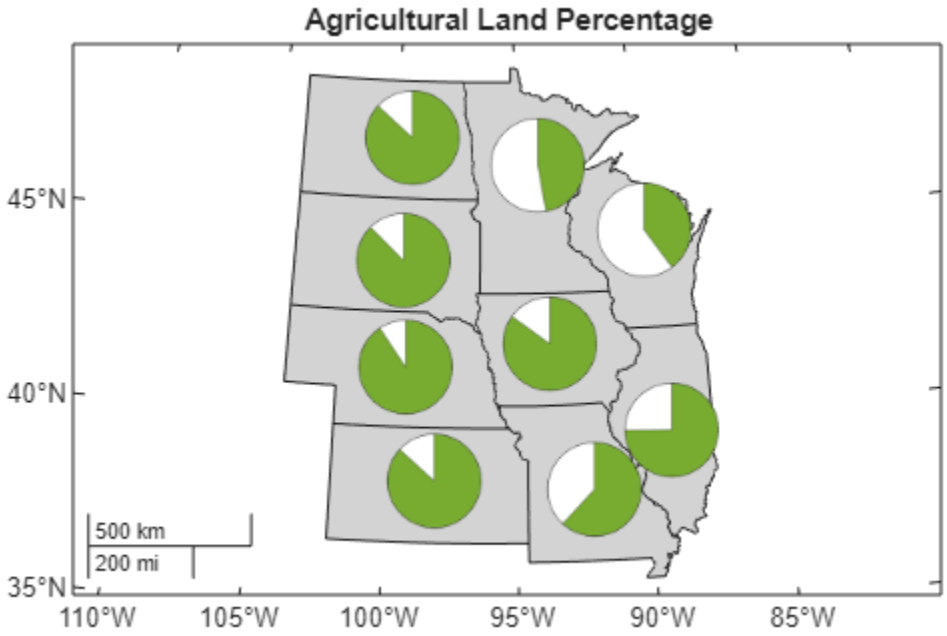
Display the agricultural land percentage for each state by using pie chart icons. The pie charts icons were created using data from the National Agricultural Statistics Service (NASS) Farms and Land in Farms 2019 Summary [1] on page 3-52. For each state:

- Get the latitude and longitude coordinates of a point in the state.
- Specify the name of the pie chart icon for the state. The icons are contained in the folder `"state-farm-icons"`, and are included with the example as supporting files. The icon files follow the naming convention `"statename.png"`, where *statename* is the name of a state. Where applicable, the names of the icon files use hyphens instead of spaces.
- Display the icon at the latitude and longitude coordinates. The green slice of the pie chart represents the agricultural land percentage.

```
hold on
for i = 1:height(midwest)
    % get latitude and longitude coordinates
    lat = midwest.LabelLat(i);
    lon = midwest.LabelLon(i);

    % specify name of file
    filename = replace(midwest.Name(i)," ","-") + ".png";
    filepath = fullfile("state-farm-icons",filename);

    % create icon chart
    geoiconchart(lat,lon,filepath,SizeData=60);
end
```

Customize the appearance of the map by adding a title and removing the graticule.

```
title("Agricultural Land Percentage")

mx = gca;
mx.GraticuleLineStyle = "none";
```



[1] National Agricultural Statistics Service. “Number of Farms, Land in Farms, and Average Farm Size — States and United States: 2018–2019.” *In Farms and Land in Farms 2019 Summary*, 6. USDA, February 2020. https://www.nass.usda.gov/Publications/Todays_Reports/reports/fnlo0220.pdf.

Animate Movement Along Path

Use an icon to represent movement along a path. Animate the icon by incrementally updating its latitude and longitude coordinates.

Read the path of a UAV into a geospatial table. The geospatial table represents the path using point shapes in geographic coordinates. Extract the latitude and longitude coordinates from the table.

```
GT = readgeotable("sample_uavtrack.gpx",Layer="track_points");
lat = GT.Shape.Latitude;
lon = GT.Shape.Longitude;
```

Display the path using a line over a topographic basemap. Prepare to add more data to the map by setting the hold state to `on`.

```
figure
geobasemap topographic
geoplot(lat,lon)
hold on
```

Create an icon chart at the start of the path. Specify the latitude and longitude coordinates using the first elements of the latitude and longitude vectors. Prepare to update the coordinates of the icon by returning the `IconChart` object in `ic`.

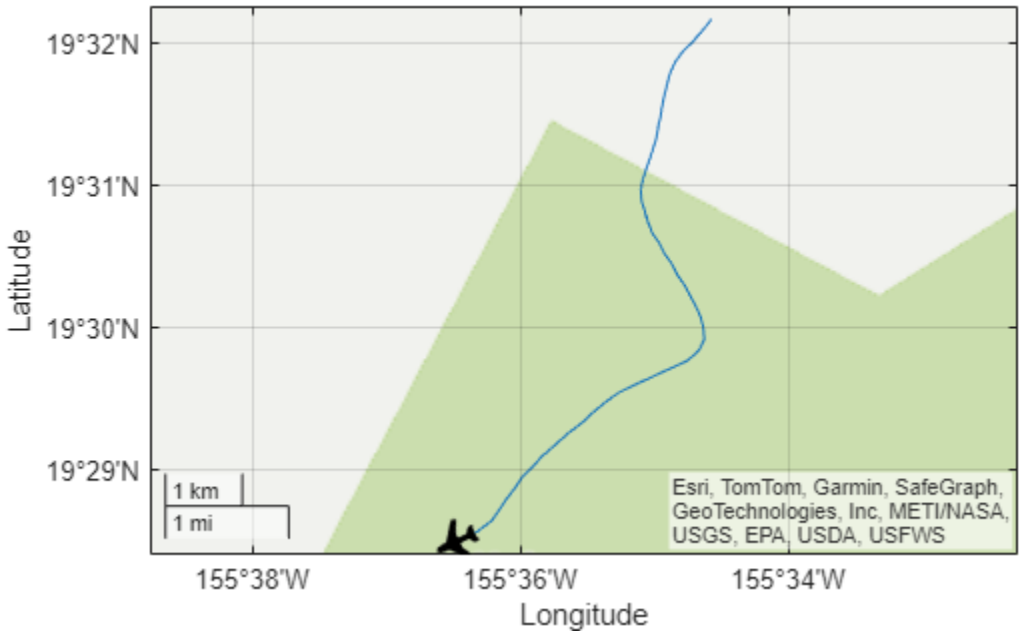
```
lat1 = lat(1);
lon1 = lon(1);
ic = geoiconchart(lat1,lon1,"uav.png");
```

Animate the icon. For each additional point in the path:

- Specify new coordinates for the icon by changing the `LatitudeData` and `LongitudeData` properties.
- Calculate a rotation angle for the icon by using the `azimuth` function. The `azimuth` function calculates angles that are measured clockwise from north.
- Specify a new rotation angle for the icon by setting the `IconRotation` property. The `IconRotation` property rotates the angle in a counterclockwise-positive direction, so use the negative of the value returned by the `azimuth` function.
- View updates to the animation.

```
gx = gca;
for i = 2:length(lat)-1
    % specify new coordinates
    ic.LatitudeData = lat(i);
    ic.LongitudeData = lon(i);

    % calculate rotation angle
    r = azimuth(lat(i-1),lon(i-1),lat(i),lon(i));

    % specify new rotation angle
```

```
    ic.IconRotation = -r;

    % view updates
    drawnow
end
```



Annotate Basemap

Use icons to add supplemental information to a basemap. For this example, use icons to label an interstate on a satellite basemap.

Read road data into the workspace as a geospatial table. The table represents the roads using line shapes in geographic coordinates.

```
roads = readgeotable("boston_roads.shp");
```

Create a subtable that contains the line shapes for Interstate 90 (I-90), which is also known as the Massachusetts Turnpike.

```
interstate = geocode("MASSACHUSETTS TURNPIKE",roads,"STREETNAME");
```

Display the line shapes for I-90 over satellite imagery.

```
figure
geobasemap satellite
hold on
geoplot(interstate,LineWidth=2,Color="#A2142F")
```

Label I-90 by adding two interstate icons. Prevent the icon from capturing mouse clicks, including clicks that add data tips, by setting the `PickableParts` property to `"none"`.

```
geoiconchart([42.3480 42.3477],[-71.0959 -71.0508],"i-90.png",SizeData=30, ...
    PickableParts="none")
```

For an example that uses OpenStreetMap® data to create a base layer with icons, see “Create Base Layer from OpenStreetMap Data” on page 4-76.

See Also

Functions
`geoiconchart` | `geoplot` | `readgeotable` | `geocode` | `exportgraphics`

Properties
IconChart Properties

Related Examples

- “Control Color and Transparency of Icon Chart” on page 3-56

Control Color and Transparency of Icon Chart

Icon charts created by the `geoiconchart` function define the icon symbol by specifying a color and a transparency for each pixel in the icon. This topic shows how to control the appearance of an icon chart by changing the color and transparency of the pixels.

In general, you change the colors of the pixels by specifying the `IconColorData` property of the `IconChart` object, and you change the transparencies of the pixels by specifying the `IconAlphaData` property of the `IconChart` object.

Remove Background from Custom Icon

When you create an icon chart by specifying the name of a custom icon file, the `geoiconchart` function sets the `IconColorData` and `IconAlphaData` properties using information that is stored in the file. If the icon file does not include transparency information, then the function displays the icon using opaque pixels. You can hide the background by changing the transparency of the corresponding pixels.

The steps in this example assume that the background color is distinct from the other icon colors.

Create Icon Chart

Create an icon chart by specifying the name of a custom icon file. Prepare to change the transparency of the icon by returning the `IconChart` object in `ic`.

```
figure
ic = geoiconchart(42.3,-71.3,"matlabicon.gif",SizeData=30);
```

Find Background Pixels

Get the icon colors by querying the `IconColorData` property of the `IconChart` object. The `IconColorData` property stores the colors using an RGB image. An RGB image is an array of RGB triplets, where each triplet defines the red, green, and blue components of one pixel.

```
iconColors = ic.IconColorData;
```

Prepare to get a colormap from the RGB image by approximating the number of colors in the icon. You can approximate the number of colors by visually inspecting the icon. This icon appears to contain six colors: white, black, orange, light orange, blue, and light blue.

```
numColors = 6;
```

Convert the RGB image to an indexed image and a colormap. Each element of the indexed image maps to a row of the colormap, and each row of the colormap is an RGB triplet that specifies the red, green, and blue components of a color. For example, the value `0` in the indexed image maps to the first color in the colormap.

```
[X,cmap] = rgb2ind(iconColors,numColors);
```

Determine which row of the colormap corresponds to the white icon background. Because the `rgb2ind` function outputs `cmap` using values of data type `double`, the red, green, and blue components of white colors are generally greater than `0.9`. Find the row of the colormap that corresponds to white.

```
whiteRowTF = cmap(:,1) > 0.9 & cmap(:,2) > 0.9 & cmap(:,3) > 0.9;
whiteRow = find(whiteRowTF)
```

```
whiteRow =
2
```

The result indicates that the second row of the colormap corresponds to white. Find the pixels in the indexed image that map to the second row of the colormap. Note that the value `1` in the indexed image maps to the second row in the colormap.

```
whitePixelTF = X == (whiteRow - 1);
whitePixels = find(whitePixelTF);
```

Change Transparency of Background Pixels

Specify the transparency of each pixel in the icon. Use `1` for opaque pixels, and use `0` for transparent pixels.

Create a matrix of `1`s. Specify the size of the matrix using the height and width of the icon, so that each element of the matrix corresponds to one pixel of the icon.

```
iconSize = size(X);
iconAlphas = ones(iconSize);
```

Set the elements of the matrix that correspond to background pixels to `0`. Then, set the `IconAlphaData` property of the icon chart to the matrix. The updated icon chart does not display the white background.

```
iconAlphas(whitePixels) = 0;
ic.IconAlphaData = iconAlphas;
```

Change Color of Pushpin Icon

By default, the `geoiconchart` function uses a pushpin icon with a dark red accent color and a light red primary color. This example shows how to change the color of the default pushpin icon. You can use a similar process to change the colors of custom icons, provided the custom icon uses a small number of colors.

Create Icon Chart

Create an icon chart using the default pushpin icon.

```
figure
ic = geoiconchart(42.3,-71.3,SizeData=40);
```

Get Icon Colors

Get the icon colors by querying the `IconColorData` property of the `IconChart` object. The `IconColorData` property stores the colors using an RGB image. An RGB image is an array of RGB triplets, where each triplet defines the red, green, and blue components for one pixel of the icon.

```
iconColors = ic.IconColorData;
```

Convert the RGB image to an indexed image and a colormap with two colors. Each element of the indexed image maps to a row of the colormap, and each row of the colormap is an RGB triplet that specifies the red, green, and blue components of a color. For example, the value `0` in the indexed image maps to the first color in the colormap.

```
[X,cmap] = rgb2ind(iconColors,2);
```

Determine the RGB triplets that correspond to the dark red accent color and the light red primary color.

- If you have Image Processing Toolbox™, you can interactively find the RGB triplets that correspond to pixels in the image by using the `impixel` function. The `impixel` function displays the icon in a figure window and waits for you to select pixels in the image. Select a pixel that uses the dark red accent color, then select a pixel that uses the light red primary color, and then press **Enter**. To use this option, specify `interactivelySelectColors` as `true`.
- If you do not have Image Processing Toolbox, or if you already know the colors that correspond to the colormap rows, you can specify the RGB triplets for each color by indexing into the colormap. The first row of the colormap corresponds to the dark red accent color, and the second row of the colormap corresponds to the light red primary color. To use this option, specify `interactivelySelectColors` as `false`.

```
interactivelySelectColors = false;
if interactivelySelectColors
    figure
```

```
    colors = impixel(X,cmap);
    accent = colors(1,:);
    primary = colors(2,:);
else
    accent = cmap(1,:);
    primary = cmap(2,:);
end
```

Specify New Colors

Specify new accent and primary colors using RGB triplets. Use black for the accent color and blue for the primary color.

```
newAccent = [0 0 0];
newPrimary = [0.3010 0.7450 0.9330];
```

Replace the colors in the colormap with the new accent and primary colors.

```
idxA = ismember(cmap,accent,"rows");
cmap(idxA,:) = newAccent;

idxP = ismember(cmap,primary,"rows");
cmap(idxP,:) = newPrimary;
```

Change Icon Color

Convert the indexed image and the new colormap to an RGB image.

```
newIconColors = ind2rgb(X,cmap);
```

Update the icon color data to use the new icon colors.

```
ic.IconColorData = newIconColors;
```

See Also

Functions
`geoiconchart` | `ind2rgb` | `find` | `ismember` | `impixel` | `validatecolor`

Properties
IconChart Properties

Related Examples

- “Create Maps Using Icons” on page 3-50
- “Working with Image Types in MATLAB”

Summary of Polygon Display Functions

The following table lists the available Mapping Toolbox patch and polygon display functions.

Function	Used For
`geoplot`	Plot 2-D points, lines, and polygons on map axes and geographic axes
`geoshow`	Display map latitude and longitude data in 2-D on `axesm`-based maps
`mapshow`	Display map data without projection in 2-D on axes
`patchm`	Patch objects projected on `axesm`-based maps
`patchesm`	Patches projected as individual objects on `axesm`-based maps
`fillm`	Filled 2-D map patches on `axesm`-based maps
`fill3m`	Filled 3-D map patches in 3-D space on `axesm`-based maps

geoplot

The `geoplot` function displays point, line, and polygon data on map axes and geographic axes. This function accepts geospatial tables and shape objects in any supported projected or geographic coordinate reference system (CRS), and accepts coordinate vectors in geographic coordinates. Unlike the other functions listed on this page, the `geoplot` function creates a `Polygon` object when you display polygon data.

geoshow and mapshow

The `geoshow` and `mapshow` functions provide a superset of functionality for displaying geographic (latitude-longitude) and projected (*xy*) geospatial data, respectively, in two dimensions. These functions accept geospatial tables, geographic data structures (`geostructs` and `mapstructs`), and coordinate vector arrays, but can also directly read shapefiles and spatially referenced raster files. To control how MATLAB renders the data, create symbol specifications by using the `makesymbolspec` function.

patchm, patchesm, fillm, and fill3m

Mapping Toolbox provides two low-level functions that display patches on `axesm`-based maps: `patchm` and `patchesm`. The `patchm` function creates one displayed object that can contain multiple unconnected faces. Unlike MATLAB patch display functions, which do not support unconnected patch faces, `patchm` separates unconnected faces using `NaN` values. Alternatively, the `patchesm` function treats each face as a separate object and returns an array of patch objects. In general, `patchm` requires more memory but is faster than `patchesm`. The `patchesm` function is useful if you need to manipulate the appearance of individual patches (as thematic maps often require).

The `fillm` and `fill3m` functions make use of the `patchm` function.

See Also

Properties
Patch Properties | Polygon Properties

More About

- “Choose a 2-D Map Display” on page 3-2
- “Create and Display Polygons” on page 6-17

Types of Data Grids and Raster Display Functions

Mapping Toolbox functions and GUIs display both regular and geolocated data grids originating in a variety of formats. Recall that regular data grids require a *referencing vector or matrix* that describes the sampling and location of the data points, while geolocated data grids require matrices of latitude and longitude coordinates.

The data grid display functions are geographic analogies to the MATLAB surface drawing functions, but operate specifically on `axesm`-based maps. Like the line-plotting functions discussed in the previous chapter, some Mapping Toolbox grid function names correspond to their MATLAB counterparts with an `m` appended.

Note Mapping Toolbox functions beginning with `mesh` are used for regular data grids, while those beginning with `surf` are reserved for geolocated data grids. This usage differs from the MATLAB definition; `mesh` plots are used for colored wire-frame views of the surface, while `surf` displays colored faceted surfaces.

Surface map objects can be displayed in a variety of different ways. You can assign colors from the figure colormap to surfaces according to the values of their data. You can also display images where the matrix data consists of indices into a colormap or display the matrix as a three-dimensional surface, with the *z*-coordinates given by the map matrix. You can use monochrome surfaces that reflect a pseudo-light source, thereby producing a three-dimensional, shaded relief model of the surface. Finally, you can use a combination of color and light shading to create a lighted shaded relief map.

The following table lists the available Mapping Toolbox surface map display functions.

Function	Used For
`geoshow`	Display map latitude and longitude data in 2-D
`mapshow`	Display map data without projection in 2-D
`meshm`	Project regular data grid on `axesm`-based map
`surfm`	Project geolocated data grid on `axesm`-based map
`pcolorm`	Project regular data grid in $z = 0$ plane on `axesm`-based map
`surfacem`	Project and add geolocated data grid to `axesm`-based map
`surflm`	3-D shaded surface with lighting on `axesm`-based map
`meshlsrm`	3-D lighted shaded relief of regular data grid on `axesm`-based map
`surflsrm`	3-D lighted shaded relief of geolocated data grid on `axesm`-based map

Fit Gridded Data to the Graticule

The toolbox projects surface objects in a manner similar to the traditional methods of map making. A cartographer first lays out a grid of meridians and parallels called the *graticule*. Each graticule cell is a geographic quadrangle. The cartographer calculates or interpolates the appropriate *x*-*y* locations for every vertex in the graticule grid and draws the projected graticule by connecting the dots. Finally, the cartographer draws the map data freehand, attempting to account for the shape of the graticule cells, which usually change shape across the map. Similarly, the toolbox calculates the *x*-*y* locations of the four vertices of each graticule cell and warps or samples the matrix data to fit the resulting quadrilateral.

In mapping data grids using the toolbox, as in traditional cartography, the finer the mesh (analogous to using a graticule with more meridians and parallels), the greater precision the projected map display will have, at the cost of greater effort and time. The graticule in a printed map is analogous to the spacing of grid elements in a regular data grid, the Mapping Toolbox representation of which is two-element vectors of the form `[`*`number-of-parallels`*`,` *`number-of-meridians`*`]`. The graticule for geolocated data grids is similar; it is the size of the latitude and longitude coordinate matrices: *`number-of-parallels`* = `mrows-1` and *`number-of-meridians`* = `ncols-1`. However, because geolocated data grids have arbitrary cell corner locations, spacing can vary and thus their graticule is not a regular mesh.

Fit Gridded Data to Fine and Coarse Graticules

This example shows how to fit gridded data to fine and coarse graticules. The choice of graticule is a balance of speed over precision in terms of positioning the grid on the map. Typically, there is no point to specifying a mesh finer than the data resolution (in this example, 180-by-360 grid cells). In practice, it makes sense to use coarse graticules for development tasks and fine graticules for final graphics production.

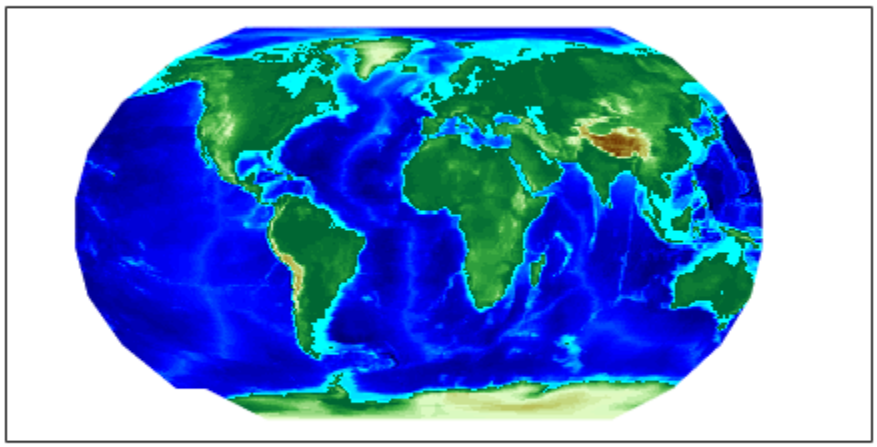
Note that, regardless of the graticule resolution, the grid data is unchanged. In this case, the data grid is a 180-by-360 matrix, and regardless of where it is positioned, the data values are unchanged.

Load elevation raster data and a geographic cells reference object.

```
load topo60c
```

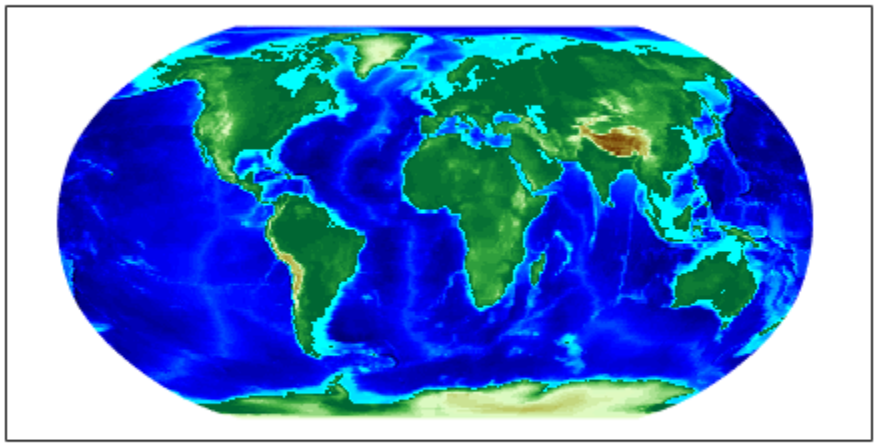
Set up a Robinson projection, specify a coarse (10-by-20) cell graticule, and display the data mapped to the graticule using a colormap appropriate for elevation data. Notice that for this coarse graticule, the edges of the map do not appear as smooth curves.

```
figure
axesm robinson
spacing = [10 20];
m = meshm(topo60c,topo60cR,spacing);
demcmap(topo60c)
```



Now reset the graticule, using the `setm` function, to make it less coarse, [50 100]. Notice that the jagged edges effect is now negligible.

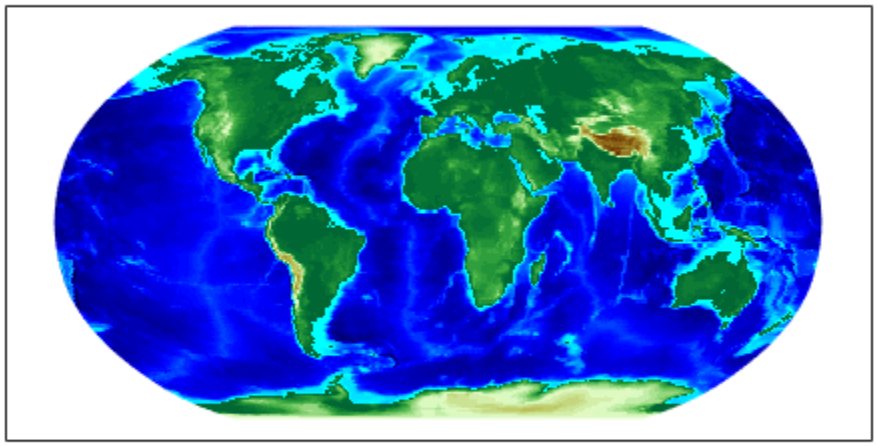
```
setm(m,'MeshGrat',[50 100])
```



Reset the graticule again, this time to a very fine grid using the `setm` function. Notice that the result does not appear to be any better than the original display with the default [50 100] graticule, but it

took much longer to produce. Making the mesh more precise is a trade-off of resolution versus time and memory usage.

```
setm(m,'MeshGrat',[200 400])
```



Create 3-D Displays with Raster Data

This example shows how to create a 3-D display of raster data on an `axesm`-based map by setting up surface views, which requires explicit horizontal coordinates. The simplest way to display raster data is to assign colors to matrix elements according to their data values and view them in two dimensions. Raster data maps also can be displayed as 3-D surfaces using the matrix values as the *z* data. The difference between regular raster data and a geolocated data grid is that each grid intersection for a geolocated grid is explicitly defined with x-y or latitude/longitude matrices or is interpolated from a graticule, while a regular matrix only implies these locations (which is why it needs a reference object).

Display Elevation Data in 3-D

Load elevation data and a geographic cells reference object for the Korean peninsula. Transform the data and reference object to a fully geolocated data grid using the `geographicGrid` function.

```
load korea5c
[lat,lon] = geographicGrid(korea5cR);
```

Next use the `km2deg` function to convert the units of elevation from meters to degrees, so they are commensurate with the latitude and longitude coordinate matrices.

```
korea5c = km2deg(korea5c/1000);
```

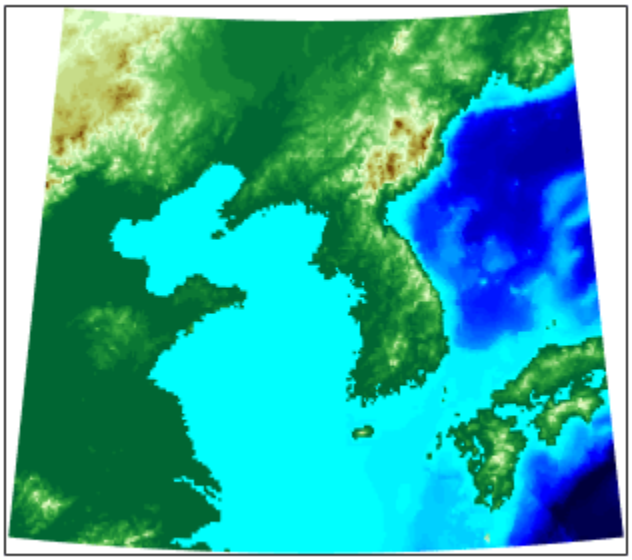
Observe the results by using the `whos` command. The `lat` and `lon` coordinate matrices form a mesh the same size as `korea5c`. This is a requirement for constructing 3-D surfaces. In `lon`, all columns contain the same number for a given row, and in `lat`, all rows contain the same number for a given column.

```
whos lat lon korea5c
```

```
  Name           Size              Bytes  Class     Attributes

  korea5c      180x240            345600  double
  lat          180x240            345600  double
  lon          180x240            345600  double
```

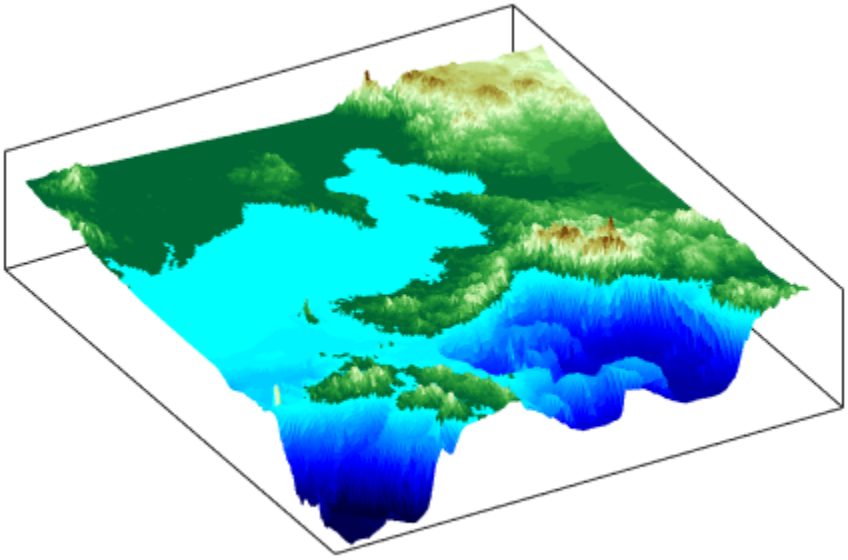
Now set up an `axesm`-based map with the equal area conic projection and, instead of using the `meshm` function to make this map, display the geolocated data grid using the `surfm` function. Set an appropriate colormap. This produces a map that is really a 3-D view seen from directly overhead (the default perspective). To appreciate that, all you need to do is to change your viewpoint.

```
axesm('MapProjection','eqaconic','MapParallels',[],...
     'MapLatLimit',[30 45],'MapLonLimit',[115 135])
surfm(lat,lon,korea5c,korea5c)
demcmap(korea5c)
tightmap
```



Specify a viewing azimuth of 60 degrees (from the east southeast) and a viewing elevation of 30 degrees above the horizon, using the `view` function.

```
view(60,30)
```



See Also

Functions
`geographicGrid` | `surfm` | `axesm`

Objects
`GeographicCellsReference`

Create Map Displays with Geographic Data

Many geospatial data sets contain geographic data, which is data referenced to latitude and longitude coordinates. This example shows how to import geographic data, display the data on a map, and customize the map. Within the example:

- Import vector and raster data files that contain geographic data.
- Create maps. Visualize vector data, raster data, and a mixture of vector and raster data on the maps.
- Customize a map by adding a scale ruler, a north arrow, and an inset map.

Import Points, Lines, and Polygons in Geographic Coordinates

A variety of data formats can store geographic vector data, including the shapefile and GPX formats. Use the `readgeotable` function to read vector data into the workspace as a geospatial table.

- `usastatehi.shp` is a shapefile that contains US states. The file represents the states using polygon shapes in geographic coordinates.
- `boston_placenames.gpx` is a GPX file that contains places in Boston, MA. The file represents the places using point shapes in geographic coordinates.
- `sample_route.gpx` is a GPX file that contains a GPS route from Boston, MA to Natick, MA. The file represents the route using a line shape in geographic coordinates.

```
states = readgeotable("usastatehi.shp");
places = readgeotable("boston_placenames.gpx");
route = readgeotable("sample_route.gpx",Layer="routes");
```

Display Polygons in Geographic Coordinates

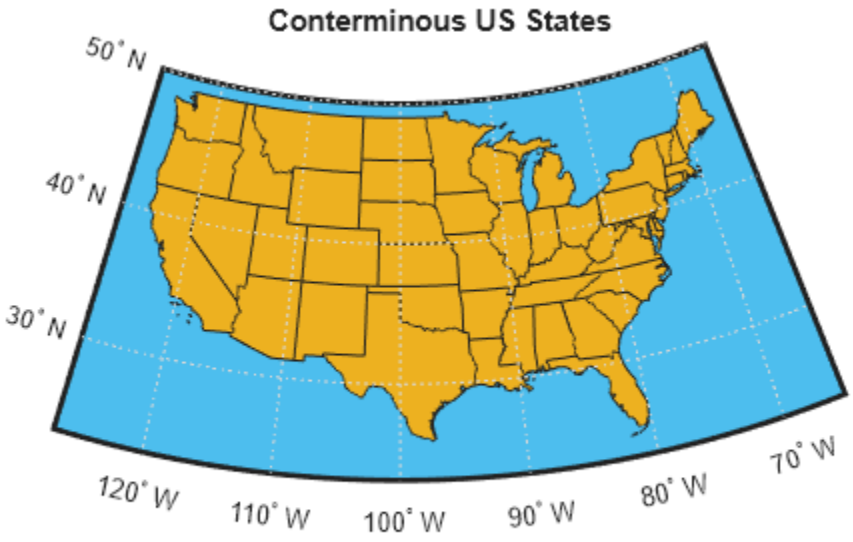
Display the shapes on an `axesm`-based map (previously referred to as map axes). `axesm`-based maps are useful for plotting geographic data in a mixture of vector and raster formats. For more information about other map display options, see “Choose a 2-D Map Display” on page 3-2.

The extent of the data is in the United States, so set up the map by using the `usamap` function. Then, project and display the shapes on the map by using the `geoshow` function. Change the background of the map to an ocean color by setting the face color of the map frame.

```
figure
ax = usamap("conus");
geoshow(states)
oceanColor = [0.3010 0.7450 0.9330];
setm(ax,"FFaceColor",oceanColor)
```

Add a title.

```
title("Conterminous US States")
```



Display Points and Lines in Geographic Coordinates

Extract the polygon shape for Massachusetts from the geospatial table of US states.

```
ma = geocode("Massachusetts",states);
```

Set up a new `axesm`-based map for a region surrounding Massachusetts. Change the background of the map to an ocean color. Provide geographic context for the map by displaying the US states in one color and Massachusetts in a second color.

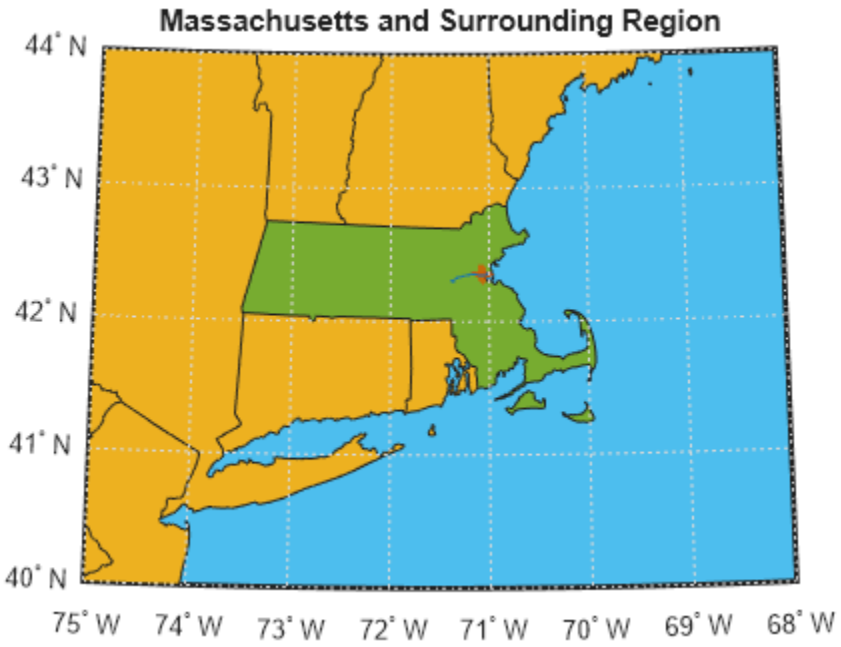
```
figure
ax = usamap("Massachusetts");
setm(ax,"FFaceColor",oceanColor)
geoshow(states)
maColor = [0.4660 0.6740 0.1880];
geoshow(ma,FaceColor=maColor)
```

Display the places and the route on the same map.

```
geoshow(places)
geoshow(route)
```

Add a title and a subtitle.

```
title("Massachusetts and Surrounding Region")
```



Specify Latitude and Longitude Limits of Map

Create two areas of interest (AOIs) from the places and route. Buffer the AOIs by 0.05 degrees. Then, calculate the latitude and longitude limits of the buffered AOIs.

```
aoiPlaces = aoiquad(places);
aoiRoute = aoiquad(route);
buffered = buffer([aoiRoute aoiPlaces],0.05);
[latlim,lonlim] = bounds(buffered,"all");
```

Create a new `axesm`-based map using the new limits.

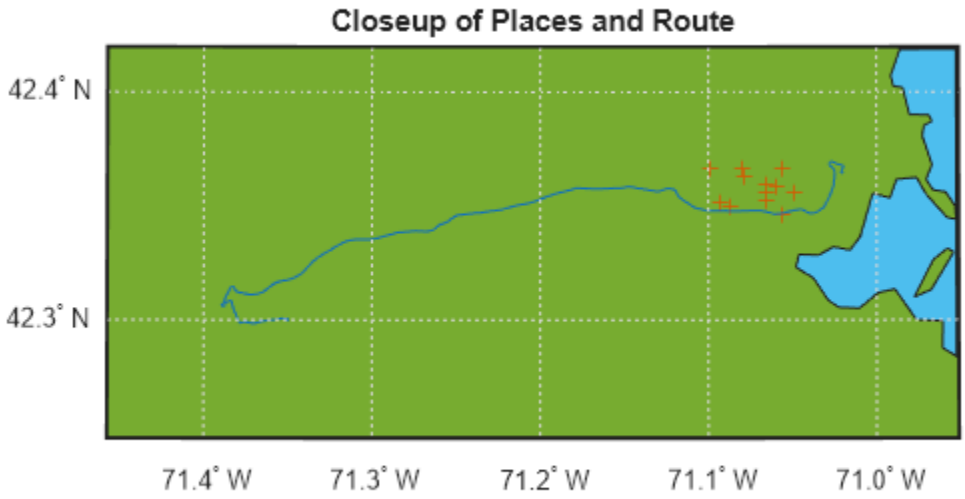
```
figure
ax = usamap(latlim,lonlim);
setm(ax,"FFaceColor",oceanColor)
```

Display the Massachusetts shape, the places, and the route on the new map.

```
geoshow(ma,FaceColor=maColor)
geoshow(places)
geoshow(route)
```

Add a title.

```
title("Closeup of Places and Route")
```



Import Raster Data in Geographic Coordinates

A variety of data formats can store geographic raster data.

- To read raster data in formats such as GeoTIFF, Esri Grid, DTED, and ENVI, use the `readgeoraster` function.
- To read an image associated with a worldfile, use the `imread` and `worldfileread` functions. The `imread` function reads the image, and the `worldfileread` function uses the worldfile to create a spatial reference object.

Import an overview image for a region surrounding Boston, Massachusetts. The coordinates of the overview image are in geographic coordinates.

```
filename = "boston_ovr.jpg";
worldfilename = getworldfilename(filename);

RGB = imread(filename);
R = worldfileread(worldfilename,"geographic",size(RGB));
```

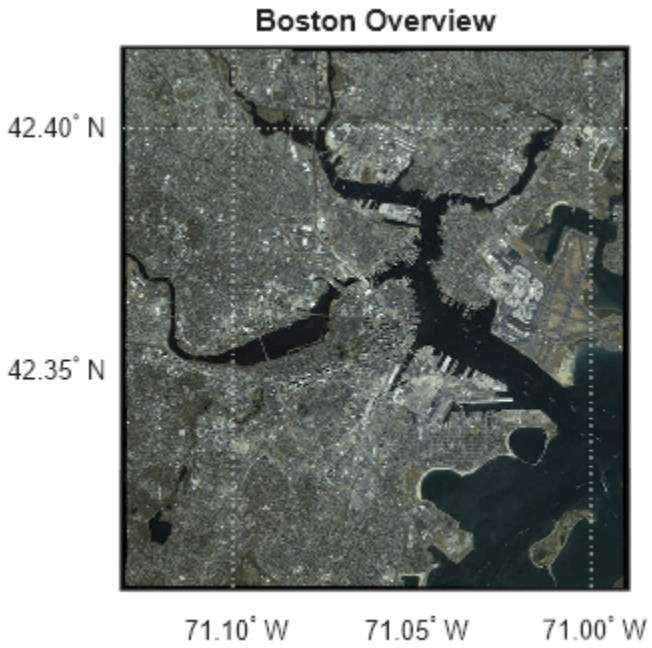
Display Raster Data in Geographic Coordinates

Create a new `axesm`-based map from the overview image and the spatial reference object. The limits of the map match the limits stored in the reference object. Customize the map by specifying the locations of the meridian and parallel labels, and by specifying the number of decimal places to include in the labels.

```
figure
ax = usamap(RGB,R);
setm(ax,"MLabelLocation",0.05,"PLabelLocation",0.05,"MLabelRound",-2,"PLabelRound",-2)
```

Display the image on the map. Add a title.

```
geoshow(RGB,R)
title("Boston Overview")
```



Display Vector and Raster Data on Same Map

Get new latitude and longitude limits from the places, the route, and the image. Create a new `axesm`-based map that uses the new latitude and longitude limits.

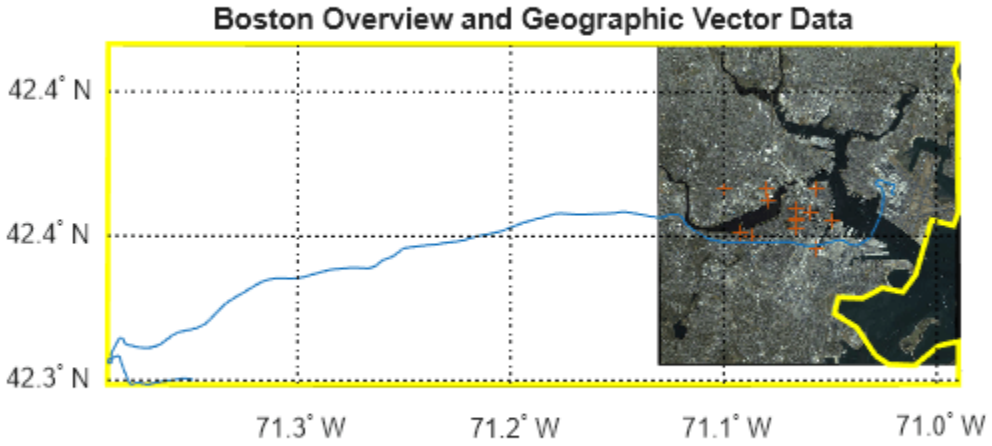
```
aoiImage = aoiquad(R);
[latlim,lonlim] = bounds([aoiPlaces aoiRoute aoiImage],"all");

figure
ax = usamap(latlim,lonlim);
setm(ax,"GColor","k","PLabelLocation",0.05,"PLineLocation",0.05)
```

Display the image, the outline of Massachusetts, the places, and the route on the new map.

```
geoshow(RGB,R)
geoshow(ma,FaceColor="none",EdgeColor="y",LineWidth=2)
geoshow(places)
geoshow(route)
```

Add a title.

```
title("Boston Overview and Geographic Vector Data")
```



Customize Map

Customize a map by adding a scale ruler, a north arrow, and an inset map.

Add Scale Ruler

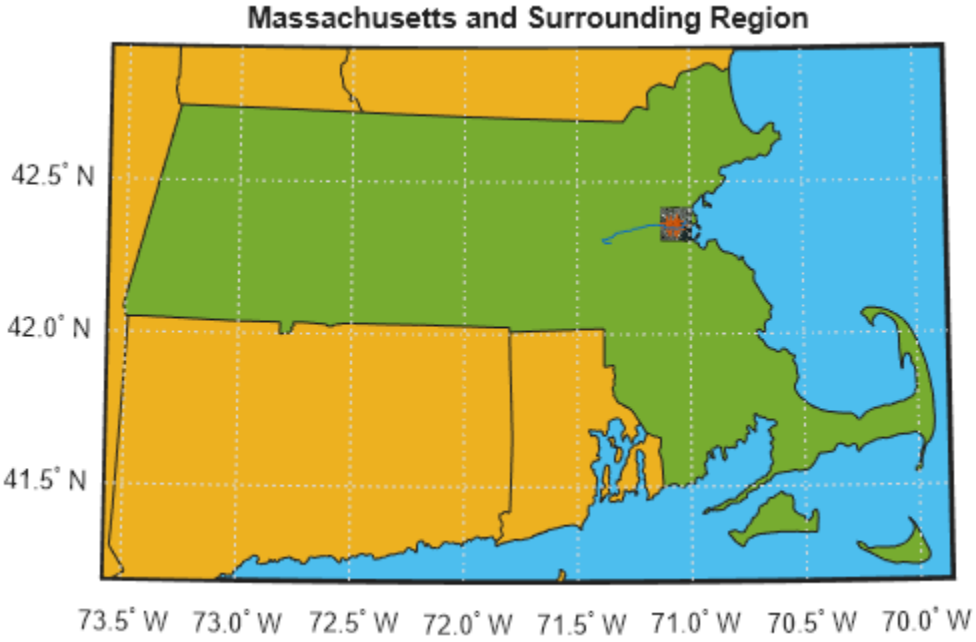
Customize a map display by including a scale ruler. A scale ruler is a graphic object that shows distances on the ground at the correct size for the projection. This example shows how to create a scale ruler that displays horizontal distances in international miles.

Buffer the Massachusetts shape by 0.05 degrees. Then, find the limits of the buffered shape.

```
bufferedMA = buffer(ma.Shape,0.05);
[latlim,lonlim] = bounds(bufferedMA);
```

Create a new map that contains the US states, Massachusetts, the overview image, the places, and the route. Add a title.

```
figure
ax = usamap(latlim,lonlim);
setm(ax,"FFaceColor",oceanColor)
geoshow(states)
geoshow(ma,FaceColor=maColor)
geoshow(RGB,R)
geoshow(places)
geoshow(route)
title("Massachusetts and Surrounding Region")
```



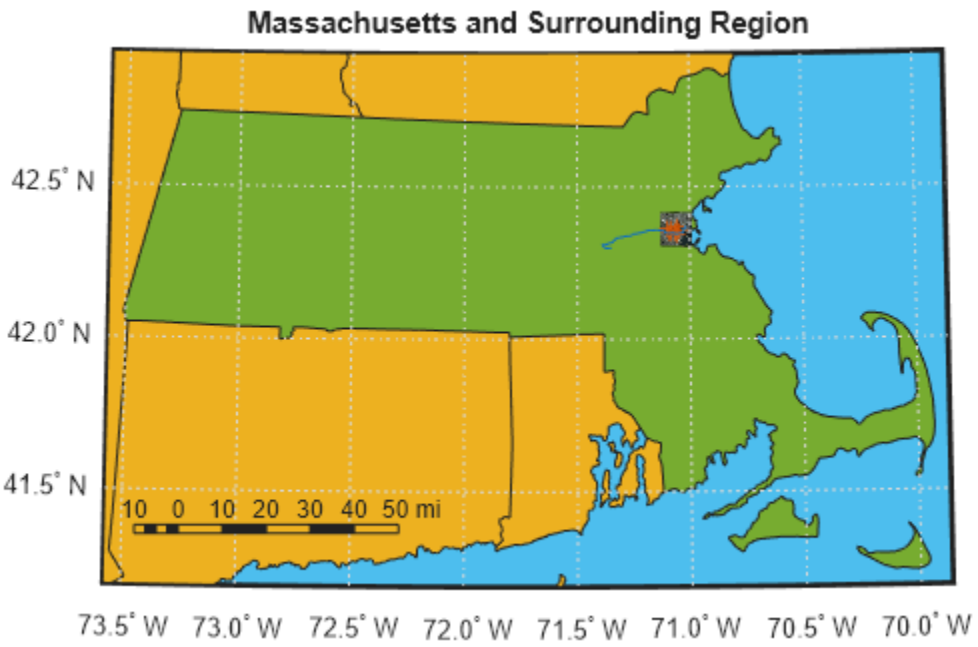
Specify a location for the scale ruler. You can interactively determine a location for the scale ruler by using the `ginput` function: `[xLoc,yLoc] = ginput(1);`

Alternatively, use a predetermined location.

```
xLoc = -127800;
yLoc = 5014700;
```

Add a scale ruler to the map.

```
scaleruler(Units="mi",RulerStyle="patches",XLoc=xLoc,YLoc=yLoc)
```

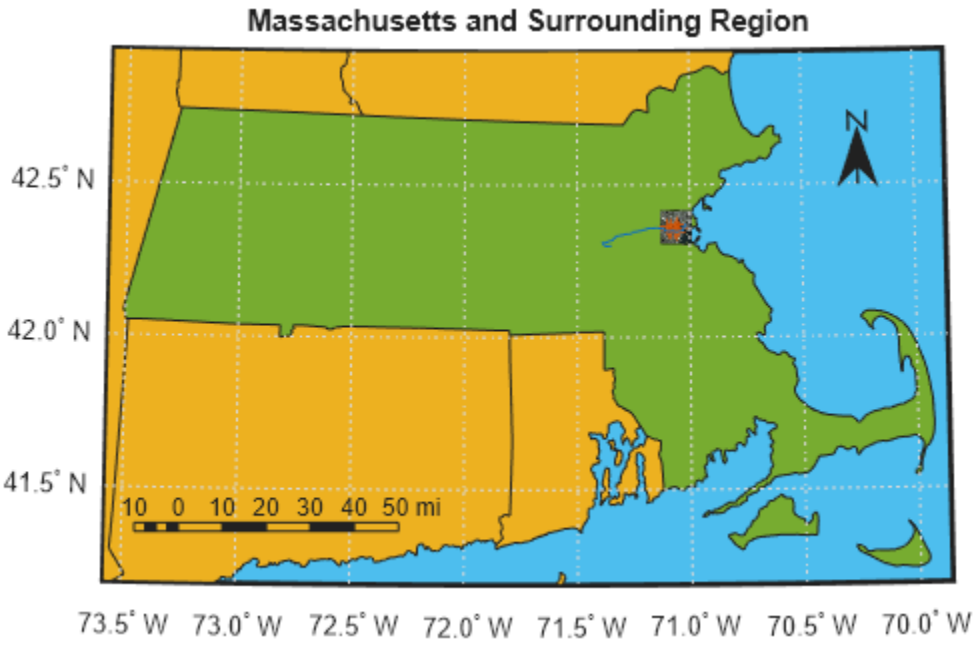


Add North Arrow

Customize the map by adding a north arrow. A north arrow is a graphic element that points to the geographic North Pole.

Specify latitude and longitude coordinates for the north arrow. Then, add the north arrow to the map.

```
northArrowLat =  42.5;
northArrowLon = -70.25;
northarrow(latitude=northArrowLat,Longitude=northArrowLon)
```

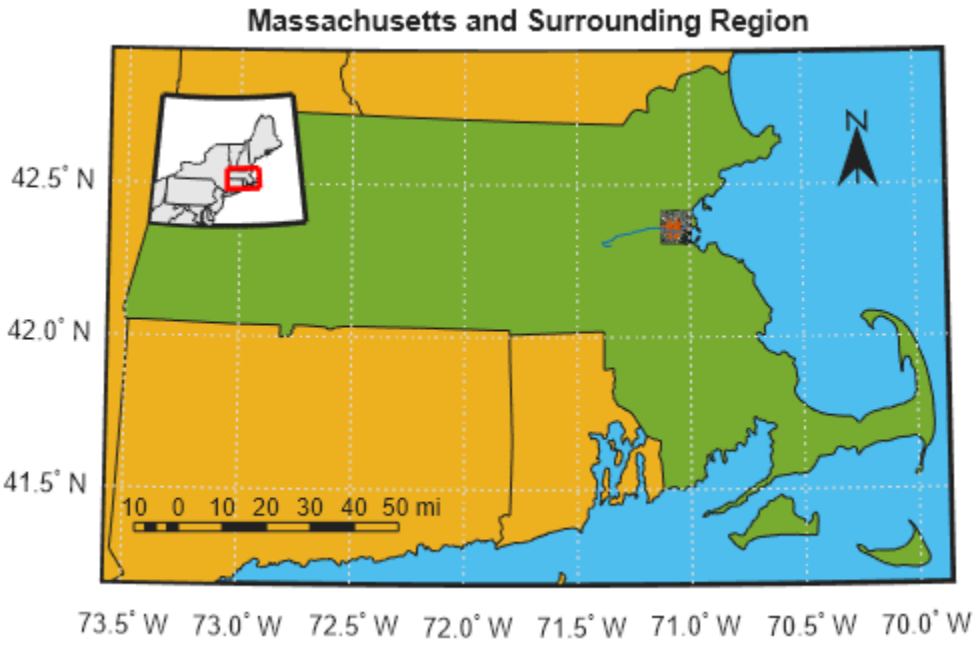


Add Inset Map

Customize the map by adding an inset map. An inset map is a smaller map displayed inside a larger map. You can use an inset map to create geographic context for the larger map. To create an inset map:

- Create an axes object in the upper-left of the map frame.
- Place a map inside the axes object. Customize the map by removing the parallel and meridian labels, setting the background color, and removing the grid lines.
- Display the US states for the surrounding region.
- Show the extent of the larger map using a red box.

```
h2 = axes(Position=[0.15 0.65 0.2 0.2],Visible="off");

usamap(["PA","ME"])
plabel off
mlabel off
gridm off
setm(h2,"FFaceColor","w")

geoshow(states,FaceColor=[0.9 0.9 0.9],Parent=h2)

mapAOI = aoiquad(latlim,lonlim);
T = table(mapAOI,VariableName="Shape");
geoshow(T,FaceColor="none",EdgeColor="r",LineWidth=2)
```



See Also

Functions
`usamap` | `geoshow` | `scaleruler` | `northarrow`

Related Examples

- “Choose a 2-D Map Display” on page 3-2
- “Supported Geospatial File Formats for Import and Export” on page 2-2

Create Maps with Data in Projected Coordinate Reference Systems

This example illustrates how to import and display geographic data that contain coordinates in a projected coordinate reference system.

In particular, this example illustrates how to

- Import specific raster and vector data sets
- Create map displays for visualizing the data
- Display multiple data sets in a map display
- Display multiple data sets with coordinates in geographic and projected coordinate reference systems in a single map display

Example 1: Import Raster Data in Projected Coordinate Reference System

Geographic raster data that contains coordinates in a projected coordinate reference system can be stored in a variety of different formats, including standard file formats such as GeoTIFF, Spatial Data Transfer Standard (SDTS), NetCDF, HDF4, or HDF5. This example illustrates importing data from a GeoTIFF file. The data in the file contains coordinates in the projected map coordinate reference system *Massachusetts State Plane Mainland Zone coordinate system*.

The coordinates of the image in the GeoTIFF file, `boston.tif`, are in a projected coordinate reference system. You can determine that by using the `geotiffinfo` function and examine the `PCS` and `Projection` field values.

```
info = geotiffinfo('boston.tif');
disp(info.PCS)
```

```
NAD83 / Massachusetts Mainland
```

```
disp(info.Projection)
```

```
SPCS83 Massachusetts Mainland zone (meter)
```

The length unit of the coordinates are defined by the `UOMLength` field in the `info` structure.

```
disp(info.UOMLength)
```

```
US survey foot
```

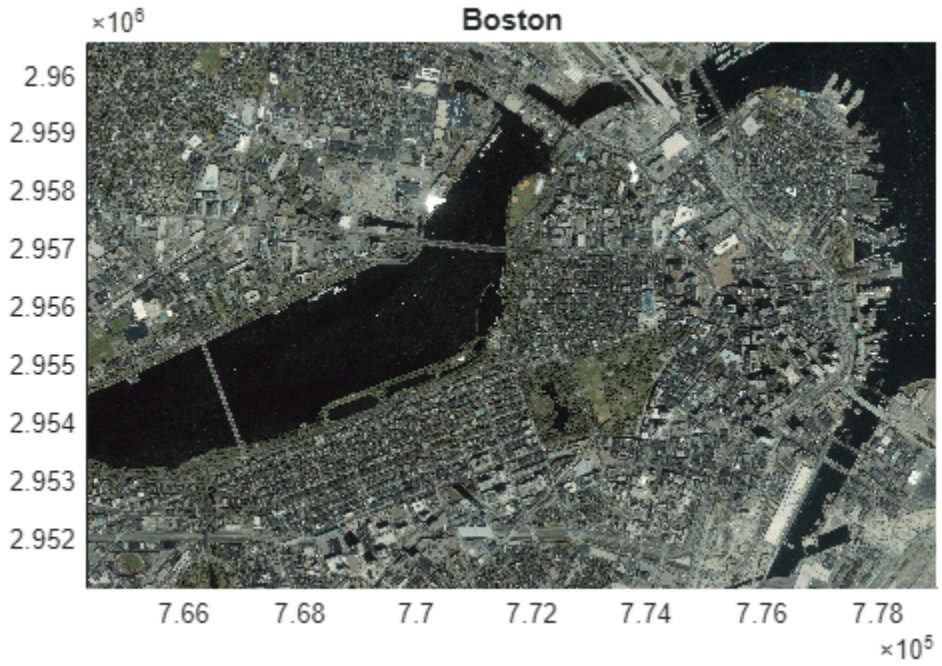
To import the image and the spatial referencing object, use `readgeoraster`.

```
[boston,R] = readgeoraster('boston.tif');
```

Example 2: Display Raster Data in Projected Coordinate Reference System

You can display the image on a regular MATLAB axes using `mapshow`, which displays the image and sets the axes limits to the limits defined by the referencing object, `R`. The coordinates, as mentioned above, are in `US survey foot` and are relative to an origin to the southwest of the map, which is why the numbers are large. The coordinates are always positive within the zone.

```
mapshow(boston,R)
axis image
title('Boston')
```



Example 3: Import Vector Data in Projected Coordinate Reference System

Geographic vector data that contains coordinates in a projected coordinate reference system can be stored in shapefiles. This example illustrates how to import vector data in a projected coordinate reference system from the shapefile, `boston_roads.shp`.

Import vector line data from the `boston_roads.shp` file as a geospatial table.

```
roads = readgeotable('boston_roads.shp');
```

The `Shape` variable of the table contains information about the line shapes. Query the projected coordinate reference system of the shapes.

```
roads.Shape.ProjectedCRS
```

```
ans =
  projcrs with properties:

                   Name: "NAD83 / Massachusetts Mainland"
          GeographicCRS: [1×1 geocrs]
       ProjectionMethod: "Lambert Conic Conformal (2SP)"
             LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Example 4: Display Vector and Raster Data in Projected Coordinate Reference System

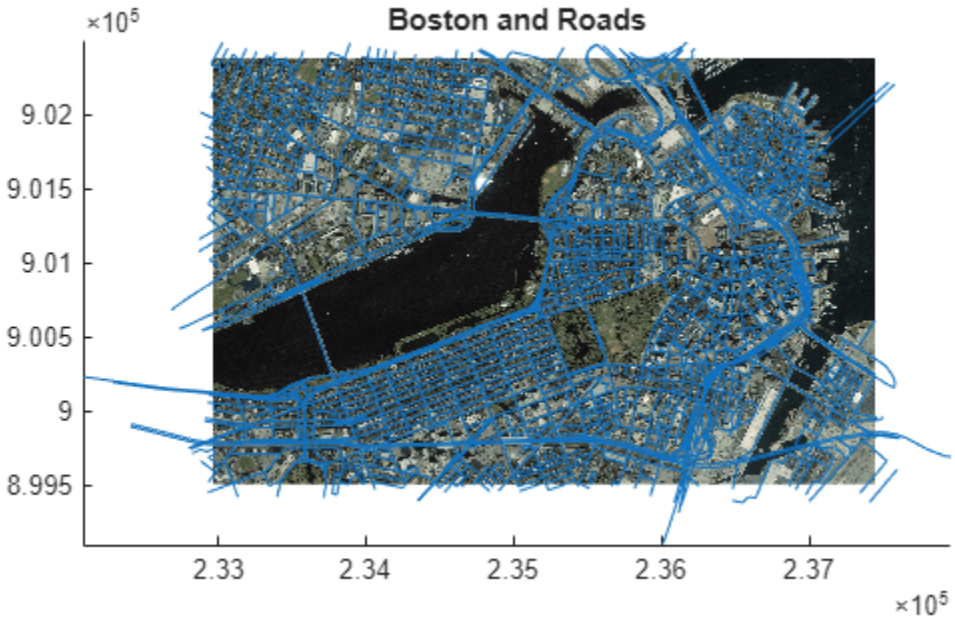
The vector and raster data in this example are in the same projected coordinate reference system. However, the vector data is in length units of meter, while the raster data is in length unit of survey foot. Convert the raster data to length units of meter and display the data on the same axes.

Convert the coordinates of the raster image from units of US survey foot to meter.

```
R.XWorldLimits = R.XWorldLimits * unitsratio('m','sf');
R.YWorldLimits = R.YWorldLimits * unitsratio('m','sf');
```

Display the raster image and vector data using `mapshow`.

```
figure
mapshow(boston,R)
mapshow(roads)
title('Boston and Roads')
```



Example 5: Display Data in both Geographic and Projected Coordinate Reference Systems

You may have geographic data whose coordinates are in latitude and longitude and other data whose coordinates are in a projected coordinate reference system. You can display these data sets in the same map display. This example illustrates how to display data in a geographic coordinate reference system (latitude and longitude) with data in a projected map coordinate reference system (Massachusetts State Plane Mainland Zone coordinate system).

Read a raster image with a worldfile whose coordinates are in latitude and longitude. Use `imread` to read the image and `worldfileread` to read the worldfile and construct a spatial referencing object.

```
filename = 'boston_ovr.jpg';
overview = imread(filename);
overviewR = worldfileread(getworldfilename(filename), 'geographic', size(overview));
```

To display the overview image and the GeoTIFF image in the same map display, you need to create a map display using a Mapping Toolbox™ projection structure containing the projection information for the data in the projected coordinate reference system, *Massachusetts State Plane Mainland Zone coordinate system*. To make a map display in this system, you can use the projection information contained in the GeoTIFF file. Use the `geotiff2mstruct` function to construct a Mapping Toolbox™ projection structure, from the contents of the GeoTIFF information structure. The `geotiff2mstruct`

function returns a projection in units of meters. Use the projection structure to define the projection parameters for the map display.

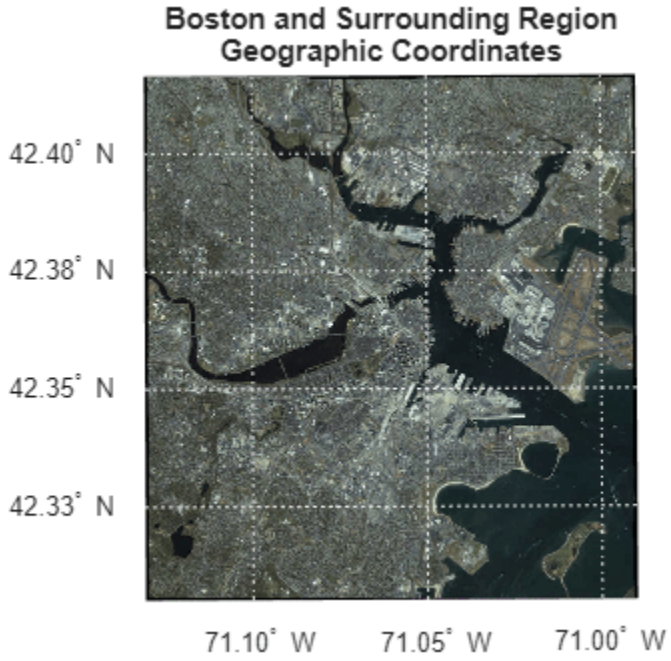
```
mstruct = geotiff2mstruct(info);
```

Use the latitude and longitude limits of the Boston overview image.

```
latlim = overviewR.LatitudeLimits;
lonlim = overviewR.LongitudeLimits;
```

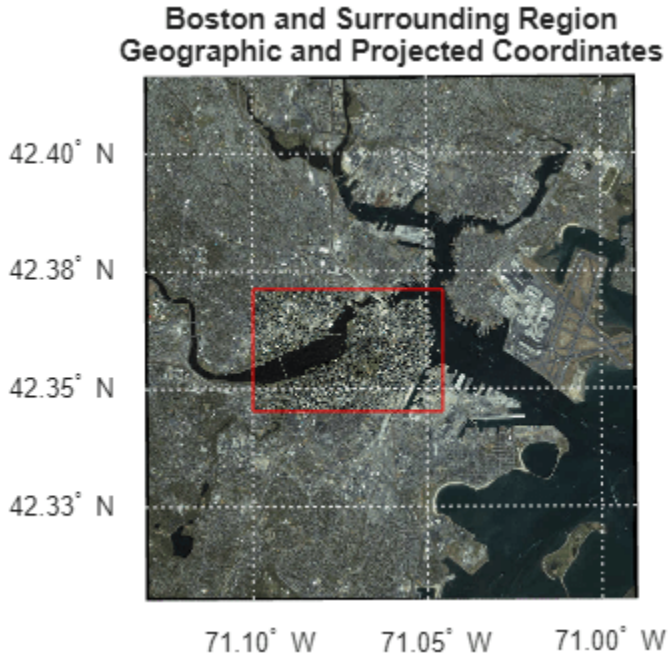
Create an `axesm`-based map by using the projection information stored in the map projection structure and set the map latitude and longitude limits. Display the geographic data in the map. `geoshow` projects the latitude and longitude coordinates.

```
figure
ax = axesm(mstruct, 'Grid', 'on',...
    'GColor', [.9 .9 .9], ...
    'MapLatlimit', latlim, 'MapLonLimit', lonlim, ...
    'ParallelLabel', 'on', 'PLabelLocation', .025, 'PlabelMeridian', 'west', ...
    'MeridianLabel', 'on', 'MlabelLocation', .05, 'MLabelParallel', 'south', ...
    'MLabelRound', -2, 'PLabelRound', -2, ...
    'PLineVisible', 'on', 'PLineLocation', .025, ...
    'MLineVisible', 'on', 'MlineLocation', .05);
geoshow(overview, overviewR)
axis off
tightmap
title({'Boston and Surrounding Region', 'Geographic Coordinates'})
```



Since the coordinates of the GeoTIFF image are in a projected coordinate reference system, use `mapshow` to overlay the more detailed Boston image onto the display. Plot the boundaries of the Boston image in red.

```
mapshow(boston, R)
plot(R.XWorldLimits([1 1 2 2 1]), R.YWorldLimits([1 2 2 1 1]), 'Color', 'red')
title({'Boston and Surrounding Region', 'Geographic and Projected Coordinates'})
```



Zoom to the geographic region of the GeoTIFF image by setting the axes limits to the limits of the Boston image and add a small buffer. Note that the buffer size (`delta`) is expressed in meters.

```
delta = 500;
xLimits = R.XWorldLimits + [-delta delta];
yLimits = R.YWorldLimits + [-delta delta];
xlim(ax,xLimits)
ylim(ax,yLimits)
setm(ax, 'Grid', 'off');
```

You can overlay the road vectors onto the map display. Use a symbol specification to give each class of road its own color.

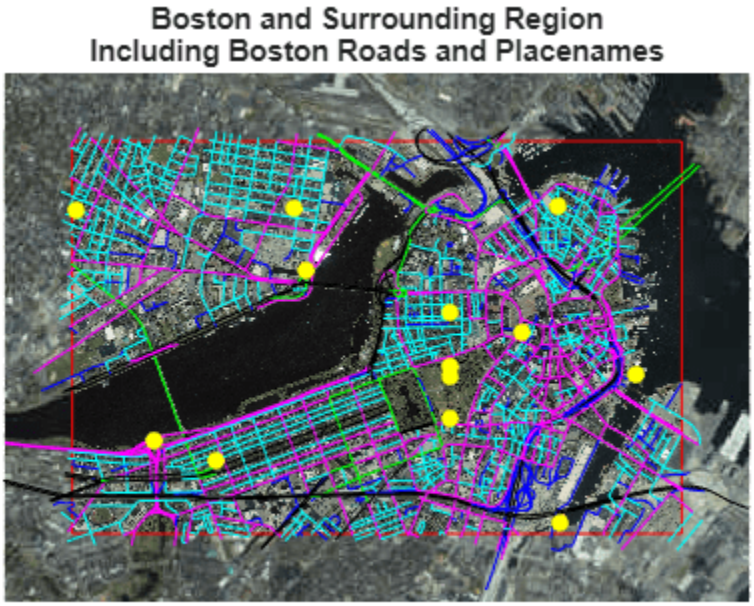
```
roadColors = makesymbolspec('Line',...
    {'CLASS', 2, 'Color', 'k'}, ...
    {'CLASS', 3, 'Color', 'g'},...
    {'CLASS', 4, 'Color', 'magenta'}, ...
    {'CLASS', 5, 'Color', 'cyan'}, ...
    {'CLASS', 6, 'Color', 'b'},...
    {'Default',  'Color', 'k'});
mapshow(roads, 'SymbolSpec', roadColors)
title({'Boston and Surrounding Region','Including Boston Roads'})
```

You can also overlay data from a GPS stored in a GPX file. Import point geographic vector data from the `boston_placenames.gpx` file included with the Mapping Toolbox™ software. The file contains latitude and longitude coordinates of geographic point features in part of Boston, Massachusetts, USA.

```
places = readgeotable('boston_placenames.gpx');
```

Overlay the places onto the map and increase the marker size, change the markers to circles and set their edge and face colors to yellow.

```
geoshow(places, 'Marker','o', 'MarkerSize', 6, ...
    'MarkerEdgeColor', 'y', 'MarkerFaceColor','y')
title({'Boston and Surrounding Region','Including Boston Roads and Placenames'})
```



Data Set Information

The files `boston.tif` and `boston_ovr.jpg` include materials copyrighted by GeoEye, all rights reserved. GeoEye was merged into the DigitalGlobe corporation on January 29th, 2013. For more information about the data sets, use the commands `type boston.txt` and `type boston_ovr.txt`.

The files `boston_roads.shp` and `boston_placenames.gpx` are from the Bureau of Geographic Information (MassGIS), Commonwealth of Massachusetts, Executive Office of Technology and Security Services. For more information about the data sets, use the commands `type boston_roads.txt` and `type boston_placenames_gpx.txt`.

Interactively Identify Geographic Locations

Find the latitude-longitude coordinates of locations on a map by interactively selecting the locations.

This page shows how to create similar maps using map axes (*since R2023a*) and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Load Data

Read a shapefile containing world land areas into the workspace.

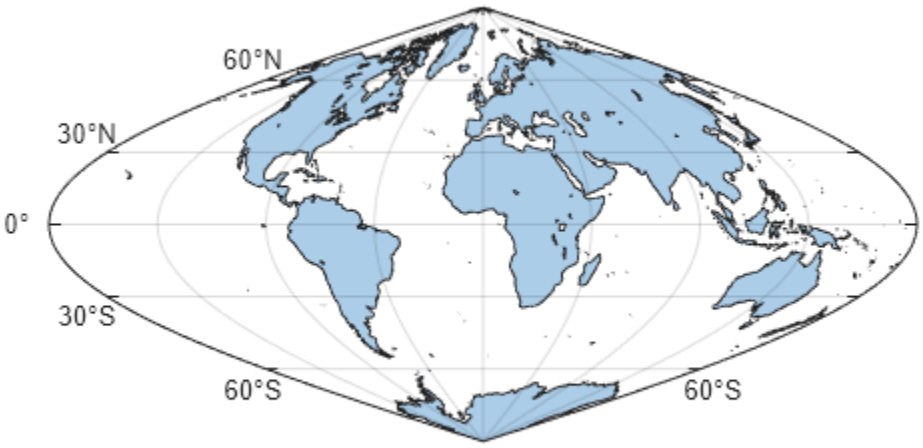
```
land = readgeotable("landareas.shp");
```

Find Coordinates on Map Axes

Interactively identify locations on map axes by using the `ginput` function.

Create a new map that uses a sinusoidal projection. Create the projected CRS using the ESRI code `54008`. Provide geographic context for the map by displaying the land areas.

```
figure
proj = projcrs(54008,Authority="ESRI");
newmap(proj)

geoplot(land)
hold on
```

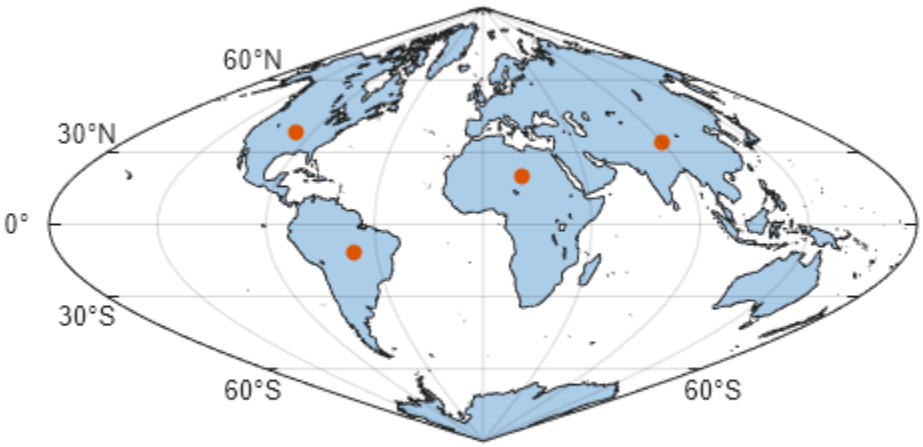


Identify the coordinates of four locations. To use predefined points, specify `interactivelySelectLocations` as `false`. To interactively select the locations, specify `interactivelySelectLocations` as `true`.

```
interactivelySelectLocations = false;
if interactivelySelectLocations
    [lat,lon] = ginput(4);
else
    lat = [38 -12 20 34];
    lon = [-98 -55 17 89];
end
```

Display the locations on the map.

```
geoscatter(lat,lon,"filled")
```



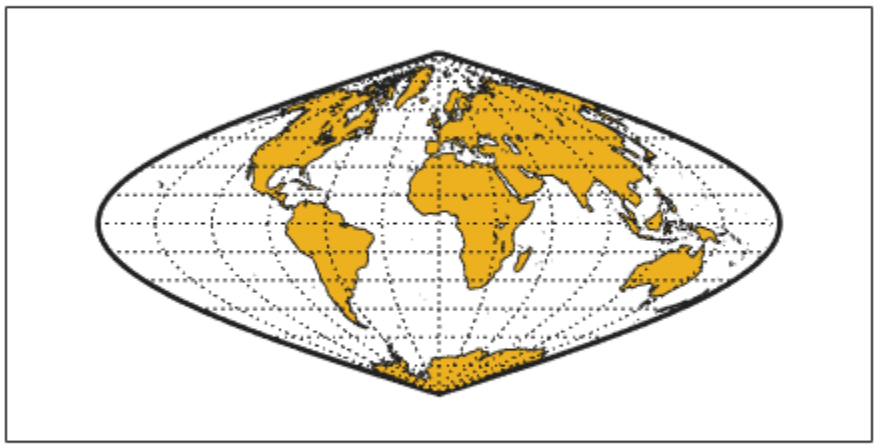
Alternatively, you can return the current mouse position on a map axes by using the `CurrentPoint` property of the axes, as in `get(gca,"CurrentPoint")`.

Find Coordinates on axesm-Based Maps

Interactively identify locations on `axesm`-based maps by using the `inputm` function.

Create an `axesm`-based map that uses a sinusoidal projection. Provide geographic context for the map by displaying the land areas.

```
figure
axesm sinusoid
framem on
gridm on
geoshow(land)
```

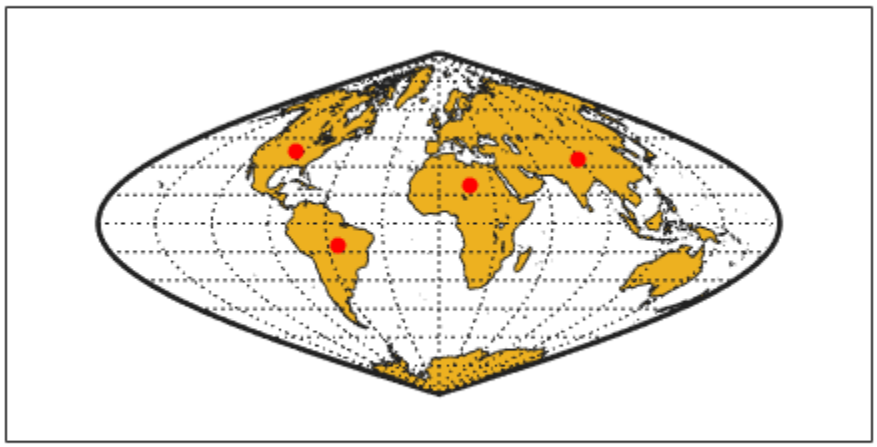


Identify the coordinates of four locations. To use predefined points, specify `interactivelySelectLocations` as `false`. To interactively select the locations, specify `interactivelySelectLocations` as `true`.

```
interactivelySelectLocations = false;
if interactivelySelectLocations
    [lat,lon] = inputm(4);
else
    lat = [38 -12 20 34];
    lon = [-96 -55 17 89];
end
```

Display the locations on the map.

```
scatterm(lat,lon,"filled","r")
```



At each location, calculate the area scale and the maximum angular deformation of the map projection. The result indicates that the `sinusoid` projection preserves area, but has variable angular distortion across the map (less distortion near the equator and more distortion near the poles).

```
[areascale,maxangdef] = distortcalc(lat,lon)
```

```
areascale = 1×4

    1.0000    1.0000    1.0000    1.0000


maxangdef = 1×4

   54.5671   11.3974    5.8094   46.9515
```

Alternatively, you can return the current mouse position on an `axesm`-based map by using the `gcpmap` function. This function is analogous to `get(gca,"CurrentPoint")`.

See Also

Functions
`newmap` | `geoscatter` | `axesm` | `scatterm` | `distortcalc`

Related Examples

- “Interactively Display Text on Maps” on page 3-93
- “Create an Interactive Map for Selecting Point Features” on page 3-112

Interactively Display Text on Maps

Display text on maps by interactively selecting the locations for the text.

This page shows how to create similar maps using map axes (*since R2023a*) and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Load Data

Read a shapefile containing polygon shapes for each of the US states into a geospatial table.

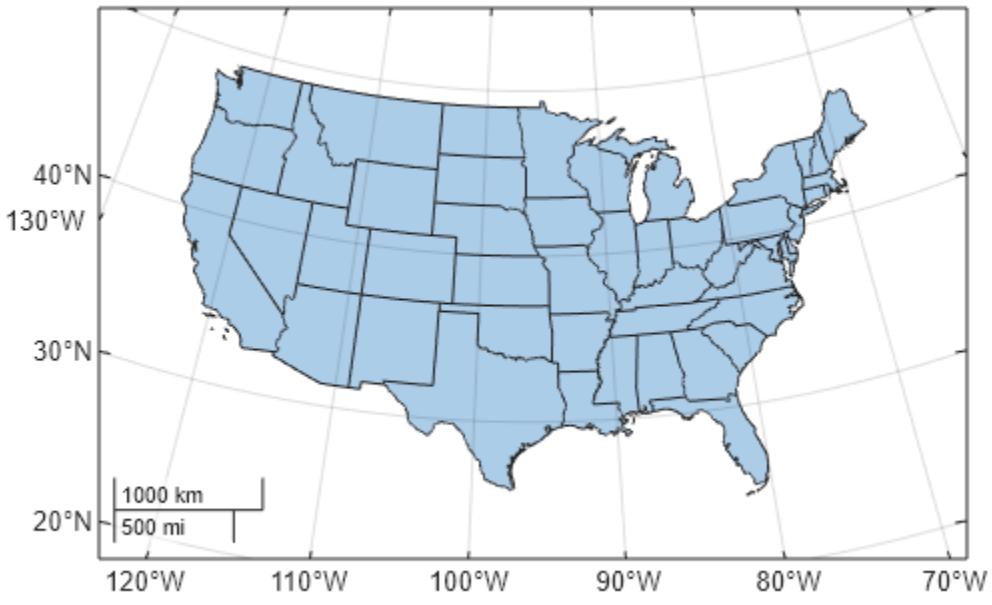
```
states = readgeotable("usastatehi.shp");
```

Display Text on Map Axes

Interactively display text on map axes by using the `gtext` function. To display text on map axes by specifying numeric arguments, use the `text` function.

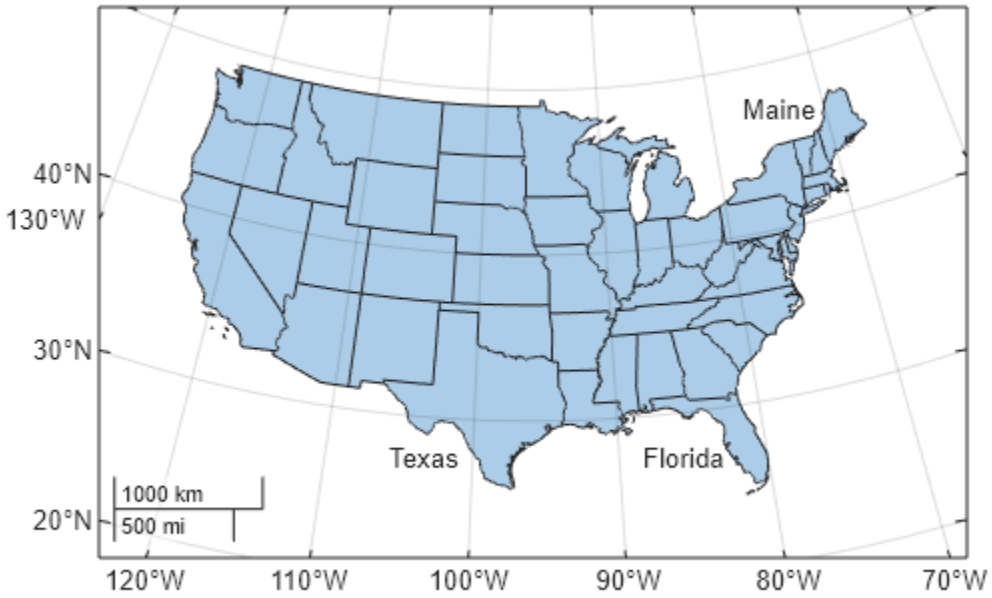
Set up a map using a projected coordinate reference system (CRS) that is appropriate for the conterminous United States. Create the projected CRS using the ESRI code `102003`.

```
figure
proj = projcrs(102003,Authority="ESRI");
newmap(proj)

geoplot(states)
hold on
```



Label Maine, Florida, and Texas by displaying text on the map. To use predefined locations, specify `interactivelySelectLocations` as `false`. To interactively select the locations, specify `interactivelySelectLocations` as `true`.

```
txt = ["Maine"; "Florida"; "Texas"];
interactivelySelectLocations = false;
if interactivelySelectLocations
    gtext(txt)
else
    lat = [47.5329 27.4865 27.2904];
    lon = [-76.9502 -88.3942 -105.6624];
    text(lat,lon,txt)
end
```



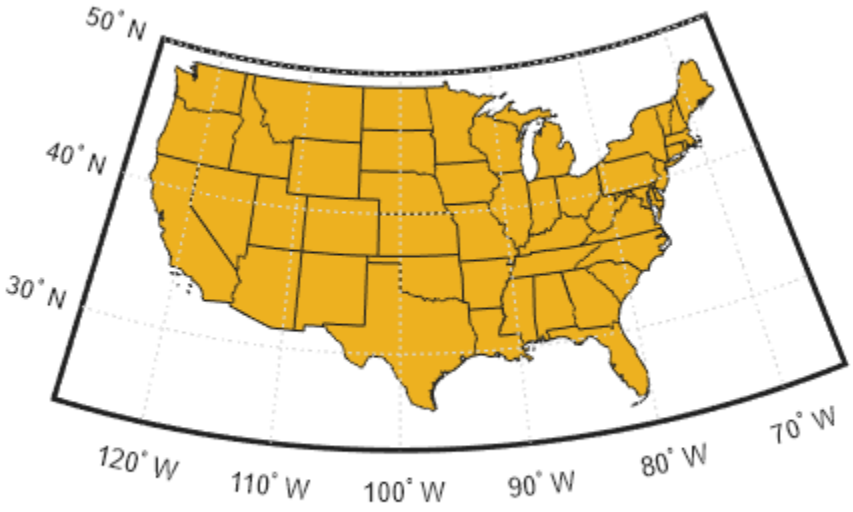
If you interactively select a location outside of the map outline, then the `gtext` function might not display the text or might display the text in an unexpected location.

Display Text on axesm-Based Maps

Interactively identify locations on `axesm`-based maps by using the `gtextm` function. To display text on `axesm`-based maps by specifying numeric arguments, use the `textm` function.

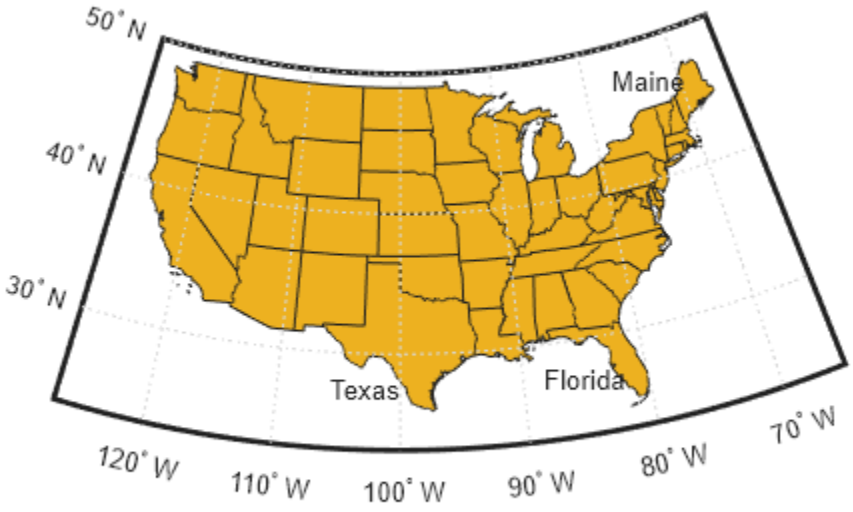
Create an `axesm`-based map that is appropriate for the conterminous United States. Provide geographic context for the map by displaying the land areas.

```
figure
usamap conus
geoshow(states)
```



Label Maine, Florida, and Texas by displaying text on the map. To use predefined locations, specify `interactivelySelectLocations` as `false`. To interactively select the locations, specify `interactivelySelectLocations` as `true`.

```
txt = ["Maine"; "Florida"; "Texas"];
interactivelySelectLocations = false;
if interactivelySelectLocations
    gtextm(txt)
else
    lat = [47.5329 27.4865 27.2904];
    lon = [-76.9502 -88.3942 -105.6624];
    textm(lat,lon,txt)
end
```



If you interactively select a location outside of the map frame, then the `gtextm` function might not display the text or might display the text in an unexpected location.

See Also

Functions
`newmap` | `geoplot` | `usamap` | `geoshow`

Related Examples

- “Interactively Identify Geographic Locations” on page 3-89
- “Add Labels and Annotations to Map Axes” on page 4-2
- “Create an Interactive Map for Selecting Point Features” on page 3-112

Add Data Tips to Point, Line, and Polygon Shapes

Create and customize data tips on point, line, and polygon shapes created using the `geoplot` function. Programmatically add data tips by using the `datatip` function, or interactively add data tips by pausing on, clicking, or tapping the shape.

Data tips appear on plotted point, line, or polygon shapes when the plotted shape meets one or more of these conditions.

- You apply color by specifying the `ColorData` or `ColorVariable` property of the plotted shape object. In this case, the data tips contain the color data.
- You change the content or appearance of the data tip by updating the `DataTipTemplate` property of the plotted shape object. In this case, the data tips reflect the changes you made to the template.
- You plot a vector of individual point shapes where the `NumPoints` property of each shape object is `1`. In this case, the data tips contain the latitude and longitude coordinates of the data points.

Programmatically Add Data Tips

Programmatically add data tips by using the `datatip` function.

Point Shapes

Read a shapefile containing world land areas into the workspace. Then, display the land areas on a map. Avoid creating data tips on the land areas by specifying the `PickableParts` name-value argument as `"none"`.

```
land = readgeotable("landareas.shp");

figure
newmap
geoplot(land,PickableParts="none")
```

Read a shapefile containing world cities into the workspace as a geospatial table. The table represents the cities using point shapes in geographic coordinates.

```
cities = readgeotable("worldcities.shp");
```

Data tips on point shapes include coordinates when each plotted shape represents an individual point. Verify that each shape represents one point by using the `ismultipoint` and `all` functions.

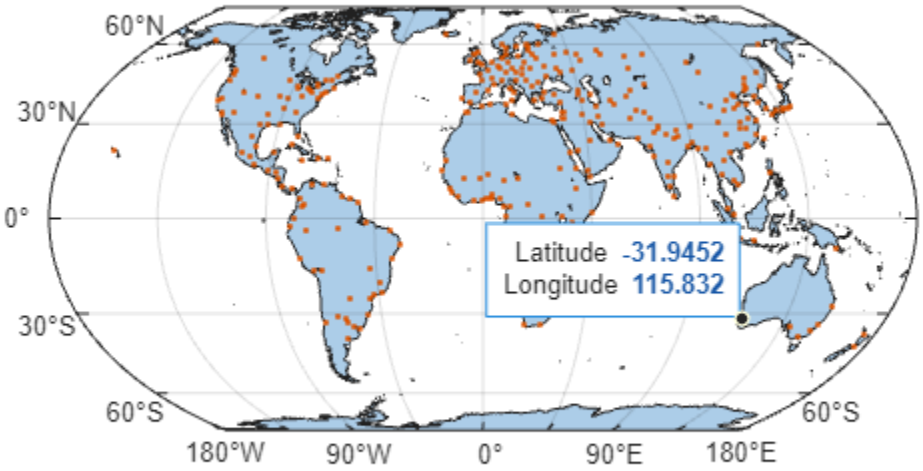
```
tf = ~ismultipoint(cities.Shape);
all(tf,"all")
```

```
ans = logical
   1
```

Display the cities on the same map. Prepare to add data tips to the points by returning the `Point` object as `pt`. Then, add a data tip by using the `datatip` function, the `Point` object, and the latitude-longitude coordinates of a city.

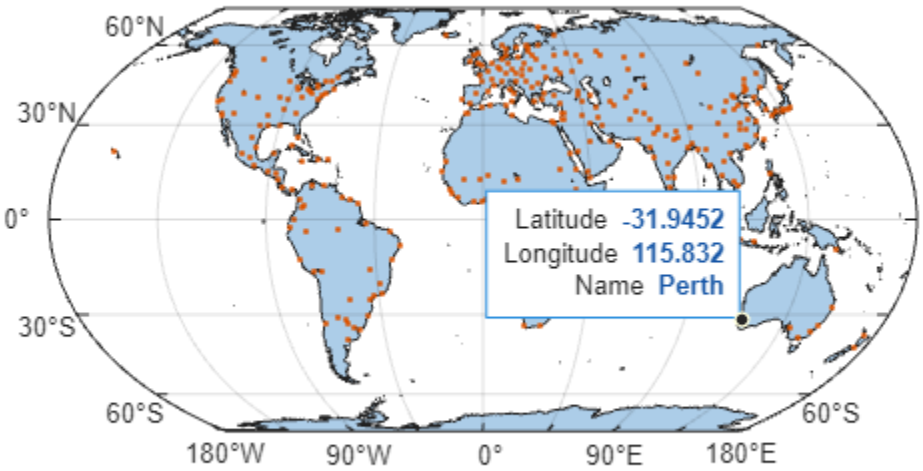
```
hold on
pt = geoplot(cities);
```

```
ptDt = datatip(pt,-31.95,115.83);
```



Add a new row to the data tip. Create the row by using the `dataTipTextRow` function. For the label, use `Name`. For the values, use the city names stored in the geospatial table. Then, add the row to the data tip by changing the `DataTipTemplate` property of the `Point` object.

```
ptDtRow = dataTipTextRow("Name",cities.Name);
pt.DataTipTemplate.DataTipRows(end+1) = ptDtRow;
```



Line Shapes

Read a shapefile containing road data into the workspace as a geospatial table. The table represents the roads using line shapes in projected coordinates.

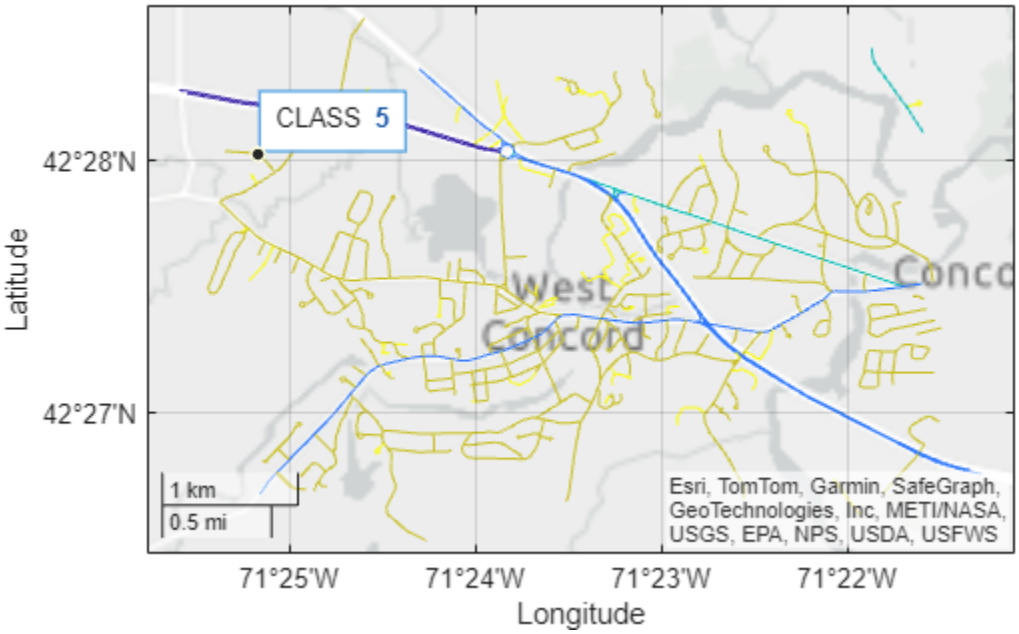
```
roads = readgeotable("concord_roads.shp");
```

Display the roads on a map. Vary the line colors according to the road type, which is stored in the `CLASS` variable of the table. Prepare to add data tips to the lines by returning the `Line` object as `pl`.

```
figure
pl = geoplot(roads,ColorVariable="CLASS");
```

Find the table row that contains a line for Craig Road. Then, add a data tip to the line by using the `datatip` function and the `DataIndex` name-value argument.

```
idxCR = find(roads.STREETNAME == "CRAIG ROAD");
plDt = datatip(pl,DataIndex=idxCR);
```

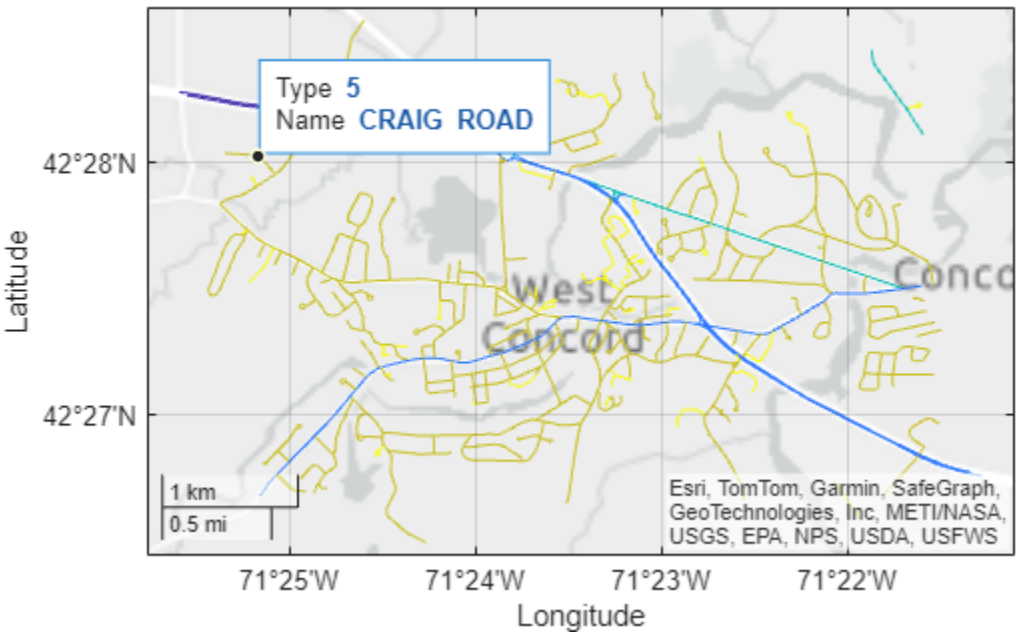


Change the data tip label from `CLASS` to `Type` by accessing the `DataTipTemplate` property of the `Line` object and setting the `Label` property.

```
pl.DataTipTemplate.DataTipRows(1).Label = "Type";
```

Add a new row to the data tip. First, create the row by using the `dataTipTextRow` function. For the label, use `Name`. For the values, use the street names stored in the geospatial table. Then, add the row to the data tip by changing the `DataTipTemplate` property of the `Line` object.

```
plDtRow = dataTipTextRow("Name",roads.STREETNAME);
pl.DataTipTemplate.DataTipRows(end+1) = plDtRow;
```



Polygon Shapes

Read a shapefile of US states into the workspace as a geospatial table. The shapefile represents the states using polygon shapes in geographic coordinates. Create a subtable that includes several midwest states: Minnesota, Iowa, Wisconsin, Illinois, Indiana, Ohio, and Michigan.

```
states = readgeotable("usastatelo.shp");
names = ["Minnesota","Iowa","Wisconsin","Illinois","Indiana","Ohio","Michigan"];
mwStates = geocode(names,states);
```

Add table variables that contain the area and perimeter of each state. Find the conversion factor from meters to kilometers by using the `unitsratio` function.

```
mToKm = unitsratio("km","m");

areaOfStates = area(mwStates.Shape)*(mToKm^2);
mwStates.Area = areaOfStates;

perimeterOfStates = perimeter(mwStates.Shape)*mToKm;
mwStates.Perimeter = perimeterOfStates;
```

Display the states on a map, and vary the polygon colors according to the area. Prepare to add data tips to the polygons by returning the `Polygon` object as `pg`.

```
figure
pg = geoplot(mwStates,ColorVariable="Area");
```

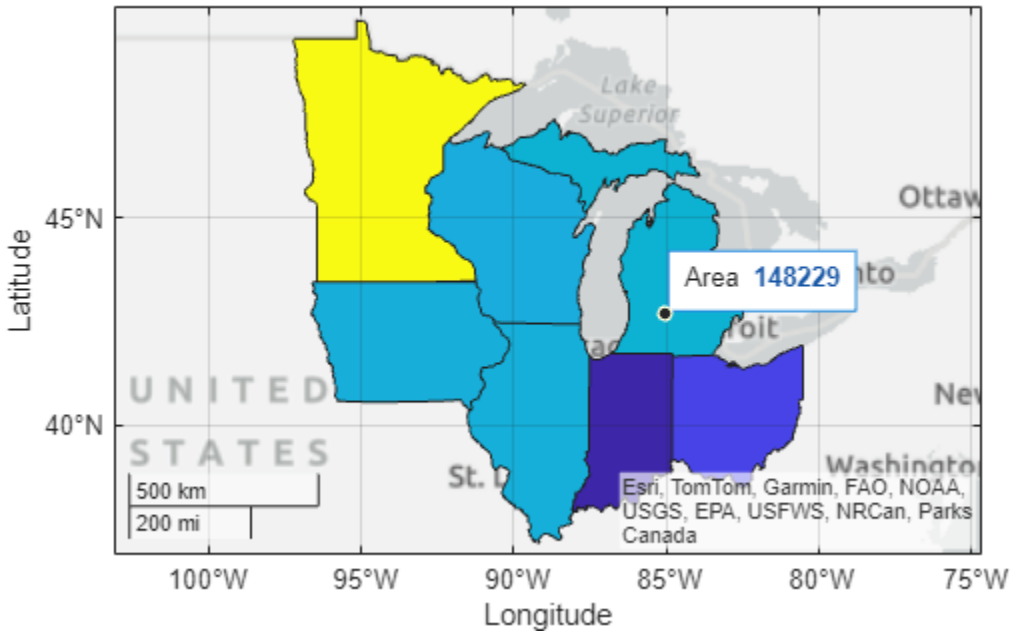
Find the table row that contains a polygon shape for Michigan. Note that the polygon contains multiple regions.

```
idxMI = find(mwStates.Name == "Michigan");
mwStates.Shape.NumRegions(idxMI)
```

```
ans =
2
```

Add a data tip to the polygon by using the `datatip` function and the `DataIndex` name-value argument. The data tip appears on the lower peninsula of Michigan.

```
pgDt1 = datatip(pg,DataIndex=idxMI);
```

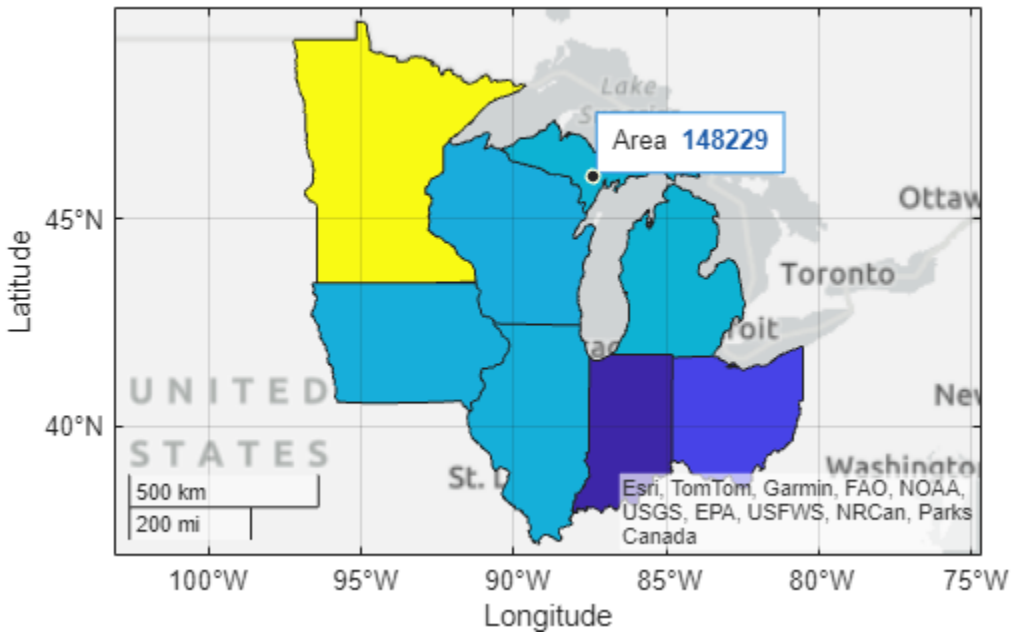


Create a data tip on the upper peninsula by specifying latitude-longitude coordinates within the peninsula and by specifying the `SnapToDataVertex` name-value argument as `"off"`.

```
pgDt2 = datatip(pg,46.15,-87.47,SnapToDataVertex="off");
```

Remove the first data tip.

```
delete(pgDt1)
```

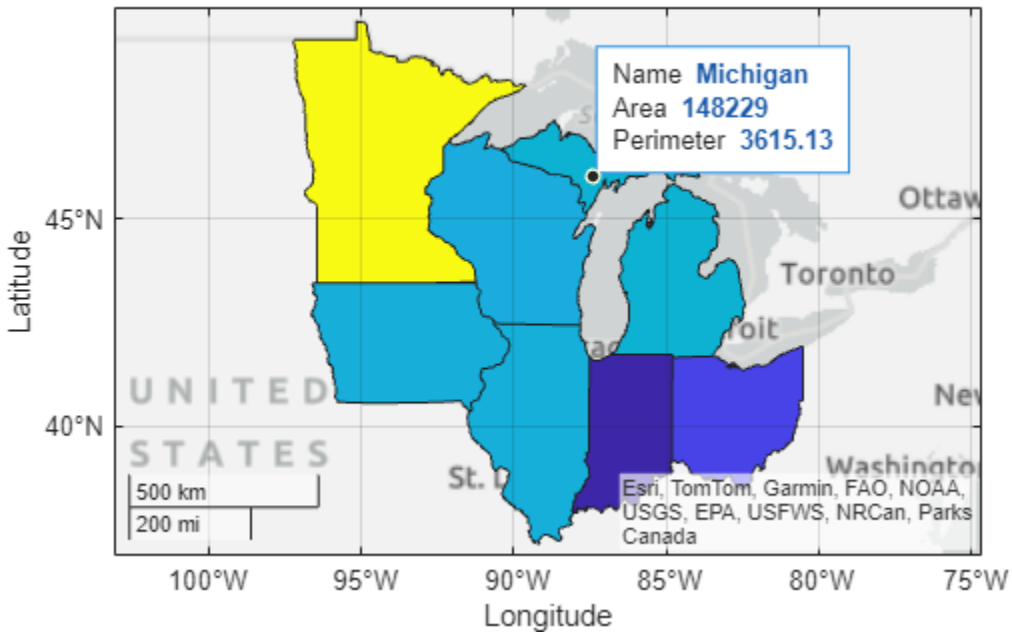


Add two new rows to the data tip. First, create the rows by using the `dataTipTextRow` function. For the labels, use `Name` and `Perimeter`. For the values, use the names and perimeters stored in the geospatial table. Then, add the rows to the data tip by changing the `DataTipTemplate` property of the `Polygon` object.

```
pgDtRows = [dataTipTextRow("Name",mwStates.Name) ...
    dataTipTextRow("Perimeter",mwStates.Perimeter)];
pg.DataTipTemplate.DataTipRows(end+1:end+2) = pgDtRows;
```

Switch the first and second rows of the data tip.

```
pg.DataTipTemplate.DataTipRows([1 2]) = pg.DataTipTemplate.DataTipRows([2 1]);
```



Interactively Add Data Tips

Interactively add data tips by pausing on, clicking, or tapping the plot.

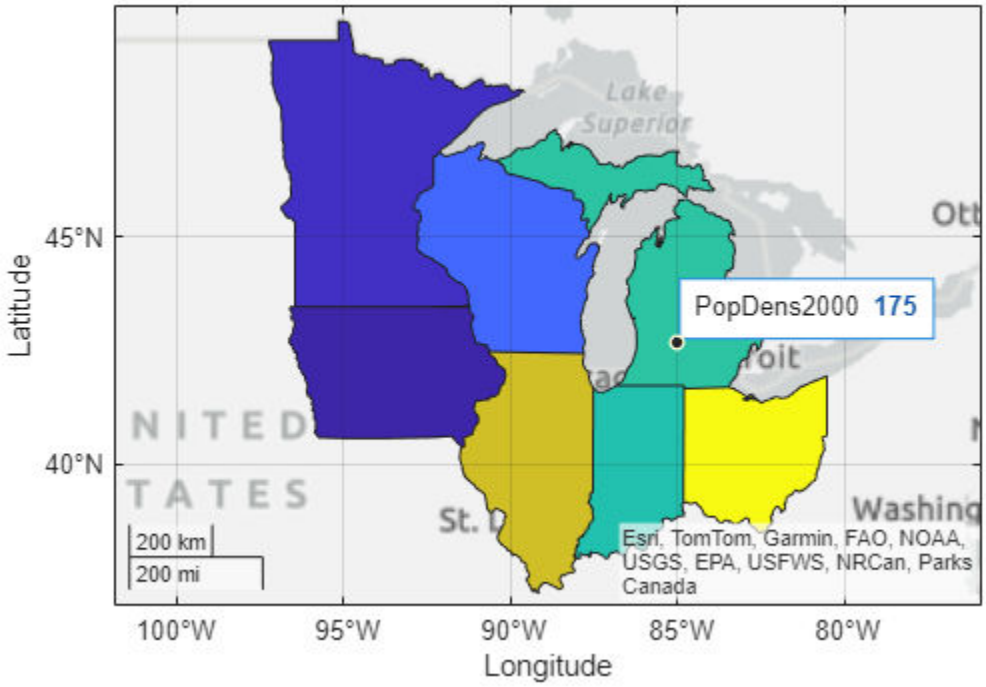
Read a shapefile of US states into the workspace as a geospatial table. The shapefile represents the states using polygon shapes in geographic coordinates. Create a subtable that includes several midwest states: Minnesota, Iowa, Wisconsin, Illinois, Indiana, Ohio, and Michigan.

```
states = readgeotable("usastatelo.shp");
names = ["Minnesota","Iowa","Wisconsin","Illinois","Indiana","Ohio","Michigan"];
mwStates = geocode(names,states);
```

Display the states on a map and vary the polygon colors according to the population density of the state in the year 2000.

```
figure
geoplot(mwStates,ColorVariable="PopDens2000")
```

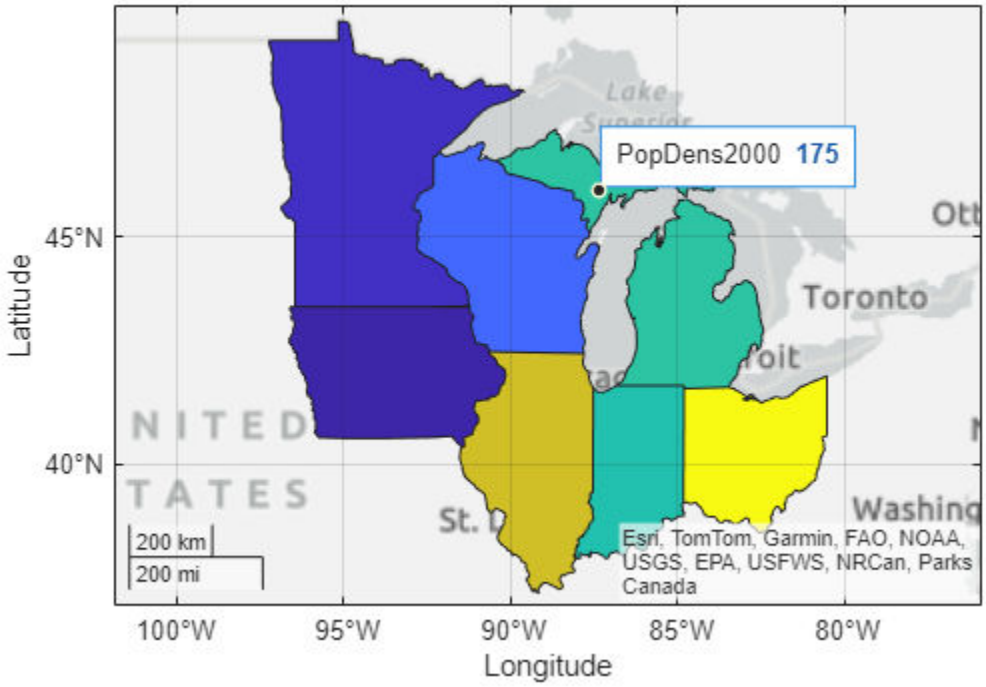
Note that the polygon shape for Michigan has two regions: a lower peninsula and an upper peninsula. Add a data tip by clicking Michigan. Regardless of the region you click, the data tip appears on the lower peninsula.



To add a data tip to a different region of the polygon, change the data tip placement method of the geographic axes object to `"interpolate"`.

```
gx = gca;
gx.InteractionOptions.DatatipsPlacementMethod = "interpolate";
```

Remove the existing data tip by clicking it. Then, add a new data tip by clicking the upper peninsula.



Tips

- In most cases, data tips for plotted shapes do not include coordinate data. Data tips on individual point shapes display geographic coordinates, even when the data originates in projected coordinates.
- Data tips for line and polygon shapes do not typically appear at the locations you specify.
  - For data tips created using the `datatip` function, you can display data tips at the closest interpolated coordinates by specifying the `SnapToDataVertex` name-value argument as `"off"`.
  - For data tips created interactively, you can display data tips at the closest interpolated location by changing the data tip placement method of the parent axes object.

    ```
    ax = gca;
    ax.InteractionOptions.DatatipsPlacementMethod = "interpolate";
    ```
- When you add data tip rows to multipoint, line, or polygon shape objects without specifying the `ColorData` or `ColorVariable` property of the object, the first row of the data tip contains the label `Row` and the index of the plotted data.
- Even when data tips do not contain content, the pointer displays a crosshair when you pause on a plotted shape.

See Also

Functions
`datatip` | `dataTipTextRow`

Properties
DataTipTemplate Properties | Point Properties | Line Properties | Polygon Properties

Related Examples

- “Create Custom Data Tips”
- “Interactively Explore and Analyze Plotted Data”
- “Control Chart Interactivity”

Work with Objects by Name

You can manipulate displayed map objects by name. Many Mapping Toolbox functions assign descriptive names to the `Tag` property of the objects they create. The `namem` and related functions allow you to control the display of groups of similarly named objects, determine the names and change them if desired, and use the name in the `set` and `get` functions.

Some mapping display functions like `framem`, `gridm`, and `contourm` assign object tags by default. You can also set the name upon display by assigning a value to the `Tag` property in mapping display functions that use property name/property value pairs. If the `Tag` does not contain a value, the name defaults to an object's `Type` property, such as `'line'` or `'text'`.

Manipulate Displayed Map Objects By Name

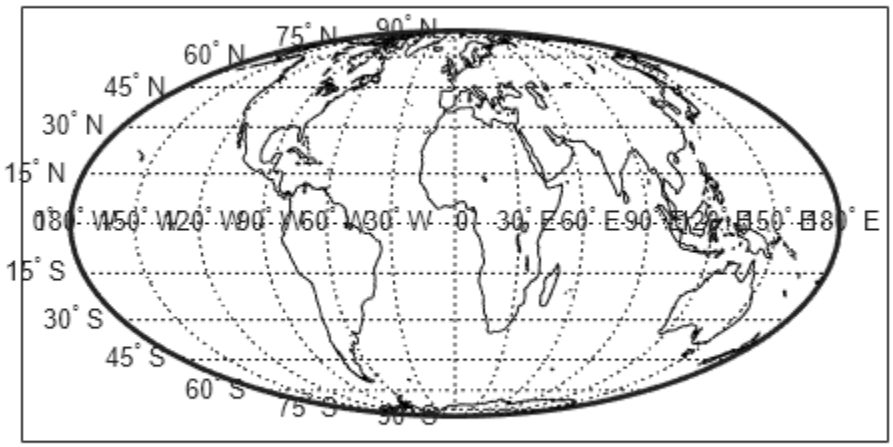
This example shows how to manipulate objects displayed on `axesm`-based maps by name. Many functions assign descriptive names to the `Tag` property of the objects they create. The `namem` and related functions allow you to control the display of groups of similarly named objects, determine the names and change them, if desired, and use the name in calls to `get` and `set`.

Display a vector map of the world.

```
f = axesm('fournier')
```

```
f =
  Axes with properties:

             XLim: [0 1]
             YLim: [0 1]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
framem on;
gridm on;
plabel on;
mlabel('MLabelParallel',0)
load coastlines
plotm(coastlat,coastlon,'k','Tag','Coastline')
```



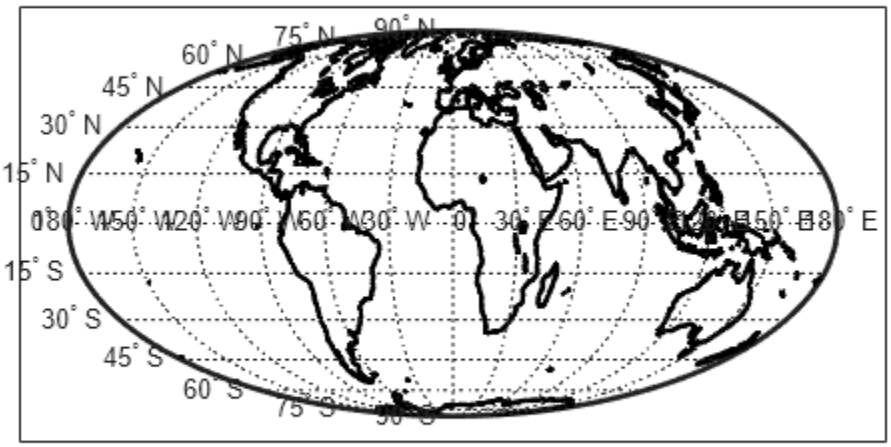
List the names of the objects in the current `axesm`-based map using `namem`.

```
namem
```

```
ans = 6×9 char array
    'PLabel   '
    'MLabel   '
    'Parallel '
    'Meridian '
    'Coastline'
    'Frame    '
```

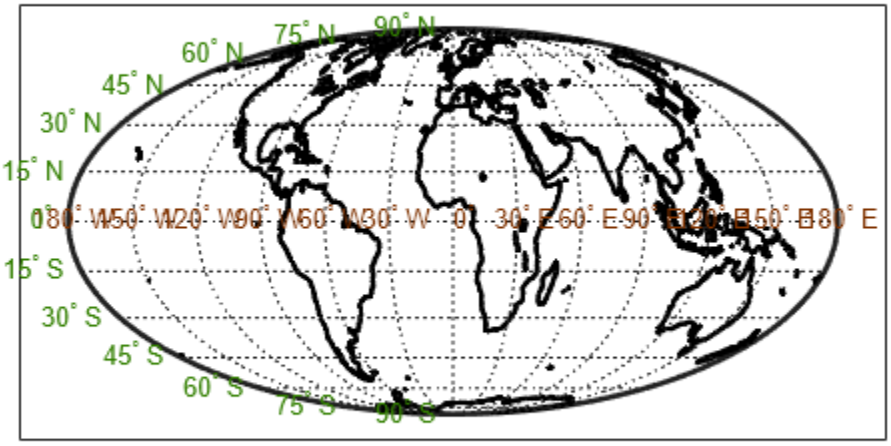
Use `handlem` to get the handles to the graphics objects displayed on an `axesm`-based map. You use these handles to get or set object properties. For example, change the line width of the coastline with `set`. If you call `handlem` with no arguments, it opens a graphical user interface that lists all the objects on the map. You can select objects interactively.

```
set(handlem('Coastline'),'LineWidth',2)
```



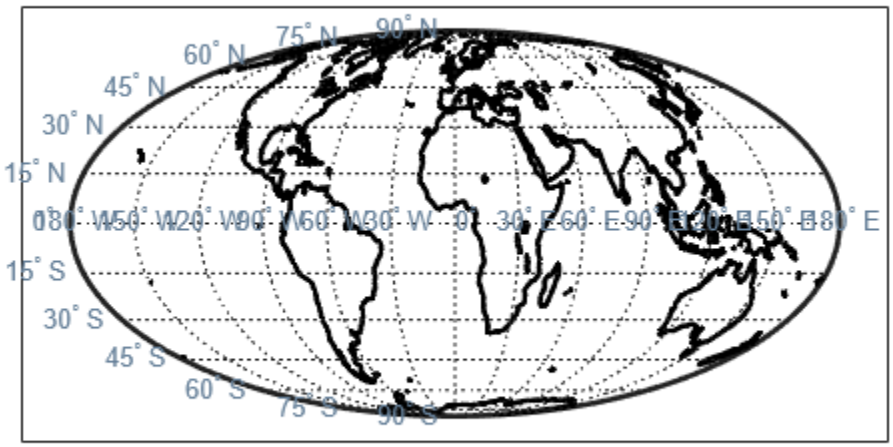
Change the colors of the meridian and parallel labels separately.

```
set(handlem('Mlabel'),'Color',[.5 .2 0])
set(handlem('Plabel'),'Color',[.2 .5 0])
```



Change the color of the labels to be the same.

```
setm(f,'fontcolor', [.4 .5 .6])
```



Use `handlem` specifying the `'all'` modifier to get a list of all text objects or all line objects.

```
t = handlem('alltext')
```

```
t =
  26×1 Text array:

  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
```

```
l = handlem('allline')
```

```
l =
  3×1 Line array:

  Line    (Parallel)
  Line    (Meridian)
  Line    (Coastline)
```

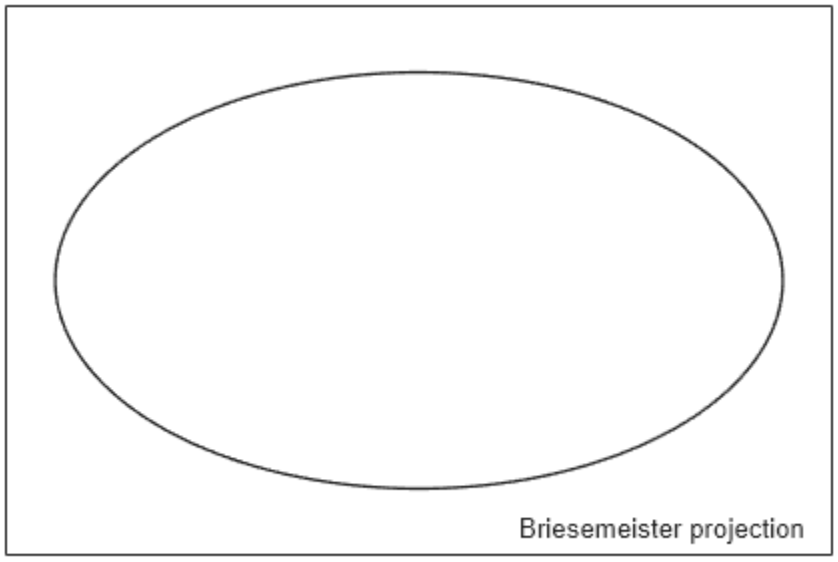
Create an Interactive Map for Selecting Point Features

This example shows how to construct a map of major world cities enhanced with coastlines and terrain. It uses the modified azimuthal Briesemeister map projection. The example includes some optional code that allows a user to interactively pick a location and get the name and location of the nearest city. To see this part of the example, you must run the complete example, pop-out the last illustration into a separate MATLAB figure, and then run the optional code at the MATLAB command line.

Step 1: Set up a Map and Render a Global Elevation Grid

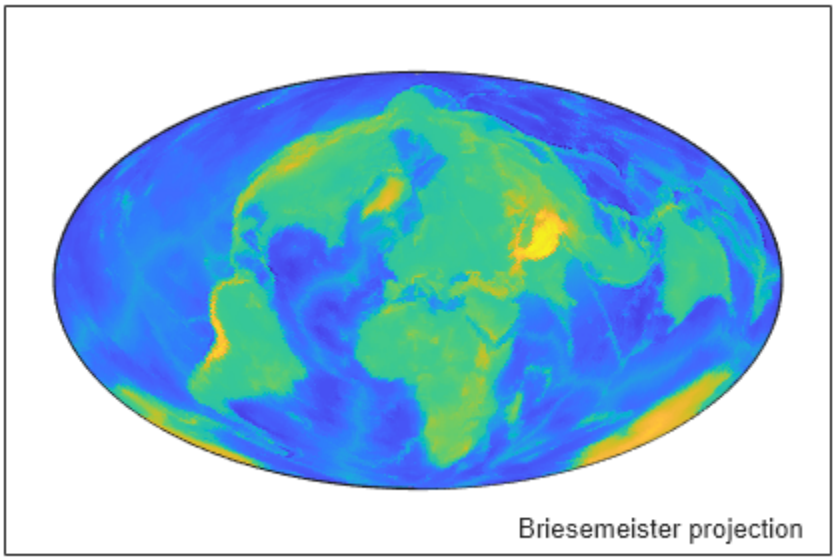
Create an `axesm`-based map.

```
figure
axesm bries
text(2.8,-1.8,'Briesemeister projection','HorizontalAlignment','right')
framem('FLineWidth',1)
```



Load elevation raster data and a geographic cells reference object. Display the data on the map.

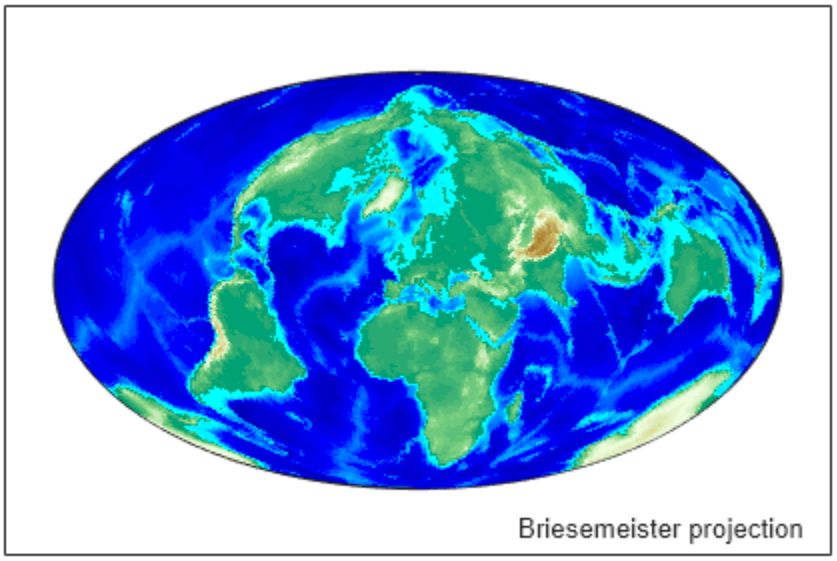
```
load topo60c
geoshow(topo60c,topo60cR,'DisplayType','texturemap')
```



Step 2: Improve the Terrain Display

Apply a colormap appropriate for elevation data. Make the display brighter.

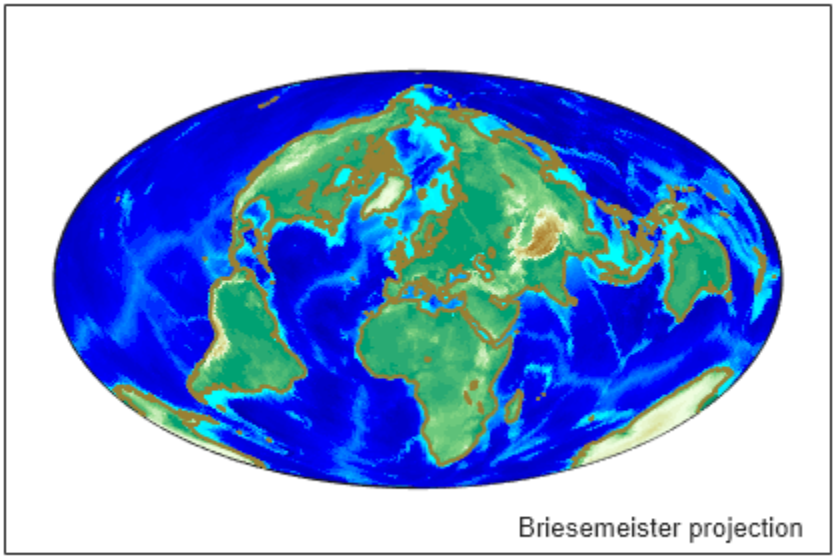
```
demcmap(topo60c)
brighten(0.5)
```



Step 3: Add Simplified Coastlines

Load global coastline coordinates. Generalize the coastlines to 0.25-degree tolerance. Then, plot the coastlines in brown.

```
load coastlines
[rlat,rlon] = reducem(coastlat,coastlon,0.25);
geoshow(rlat,rlon,'Color',[.6 .5 .2],'LineWidth',1.5)
```



Step 4: Plot City Locations with Red Point Markers

Read a shapefile containing names of cities worldwide and their coordinates in latitude and longitude.

```
cities = readgeotable('worldcities.shp');
```

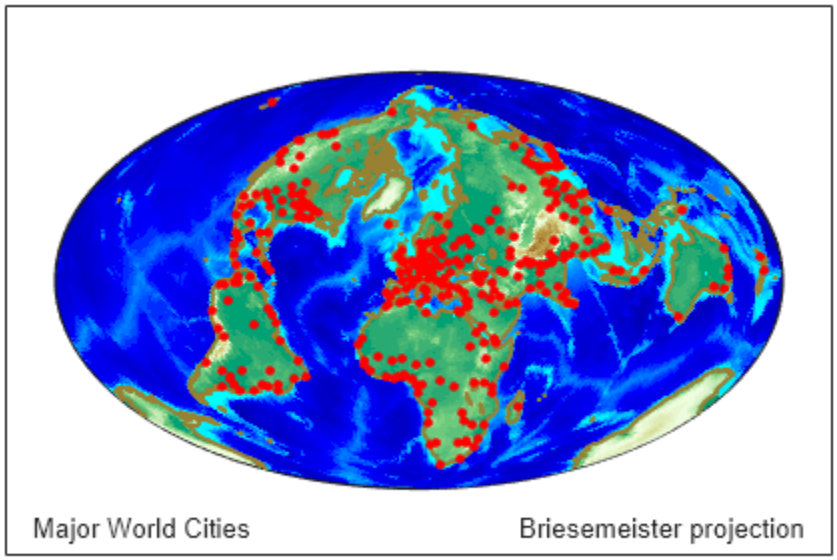
Extract the point latitudes and longitudes with `extractfield`, and add them to the map.

```
lats = cities.Shape.Latitude;
lons = cities.Shape.Longitude
```

```
lons = 318×1

   -3.9509
   54.7589
   -0.2121
   35.3894
   38.7575
  138.8528
   44.5408
   72.2474
   30.4098
    3.0397
   77.0010
   35.9214
```

```
   75.0498
    4.8892
 -149.1074
      ⋮
```

```
geoshow(lats, lons,...
        'DisplayType', 'point',...
        'Marker', 'o',...
        'MarkerEdgeColor', 'r',...
        'MarkerFaceColor', 'r',...
        'MarkerSize', 3)
text(-2.8,-1.8,'Major World Cities')
```



Step 5: Select Cities Interactively (Optional)

Now, using the map you've created, you can set up a simple loop to prompt for clicks on the map and display the name and coordinates of the nearest city. You must pop the last map you created in Step 4 into a separate MATLAB figure window, using the button that appears at the top of the map. Also, in the following code, set `runCitySelectionLoop` to `true`, and execute the code at the command line.

The code first displays text instructions at the upper left of the map. Then, it enters a loop in which it captures selected latitudes and longitudes with `inputm`. Use `distance` to calculate the great circle distance between each selected point and every city in the database. Determine index of the closest city, change the appearance of its marker symbol, and display the city's name and latitude/longitude coordinates.

```
runCitySelectionLoop = false; % Set to true to run optional city selection loop

if(runCitySelectionLoop)
    h1 = text(-2.8, 1.9, 'Click on a dot for its city name. Press ENTER to stop');
    h2 = text(-2.8, 1.7, '');
    h3 = text(-2.8, 1.5, 'City Coordinates.');
```

```
    while true
        [selected_lat,selected_lon] = inputm(1);
        if isempty(selected_lat)
            break % User typed ENTER
        end
        d = distance(lats, lons, selected_lat, selected_lon);
        k = find(d == min(d(:)),1);
        city = cities(k,:);
        geoshow(city.Shape.Latitude, city.Shape.Longitude, ...
                'DisplayType', 'point', ...
                'Marker', 'o', ...
                'MarkerEdgeColor', 'k', ...
                'MarkerFaceColor', 'y', ...
                'MarkerSize', 3)
       h2.String = city.Name;
       h3.String = num2str([city.Shape.Latitude, city.Shape.Longitude],'%10.2f');
    end
    disp('End of input.')
end
```

See Also

inputm | geoshow | demcmap | shaperead

Map Customization

- “Add Labels and Annotations to Map Axes” on page 4-2
- “Customize Appearance of Map Axes” on page 4-10
- “Create Map of Quadrangle Using Cartographic Map Layout” on page 4-16
- “The Map Frame” on page 4-23
- “Map and Frame Limits” on page 4-25
- “Map Limit Properties” on page 4-26
- “The Map Grid” on page 4-39
- “Inset Maps” on page 4-42
- “Customize Appearance of Vector Basemaps” on page 4-45
- “Create Common Plots over Basemap Images” on page 4-52
- “Change Projection of Basemap Image” on page 4-59
- “Use Basemaps in Offline Environments” on page 4-64
- “Create Base Layer from OpenStreetMap Data” on page 4-76
- “Display Data over Basemaps When Internet Access Is Intermittent” on page 4-79
- “Thematic Maps” on page 4-82
- “Create Choropleth Map of Population Density” on page 4-85
- “Create Choropleth Map from Table Data” on page 4-88
- “Create Classification Map from Table Data” on page 4-92

Add Labels and Annotations to Map Axes

Map axes objects are a type of axes object, similar to axes objects, geographic axes objects, and polar axes objects. As a result, you can use map axes with many MATLAB® graphics functions. This example shows how to convey information about your map by adding a title, a legend, text, and data tips.

By default, most plotting functions reset many of the axes properties. This example retains the plots and customizations in the axes by setting the hold state of the axes to `on`. Alternatively, you can customize the axes after plotting.

Create Map

Create a map using a projected CRS that is appropriate for Southeast Asia. Create the `projcrs` object using the ESRI code `102030`.

```
figure
p = projcrs(102030,Authority="ESRI");
newmap(p)
hold on
```

Provide geographic context for the map by displaying a subset of a shapefile containing world land areas. Avoid displaying the land areas in the legend by setting the `HandleVisibility` property to `"off"`. Display the land areas in a neutral color by setting the `SeriesIndex` property to `"none"`.

```
land = readgeotable("landareas.shp");
subland = land([1:2,5:18,20:end],:);
geoplot(subland,HandleVisibility="off",SeriesIndex="none",FaceAlpha=0.2)
```

Display world rivers using lines.

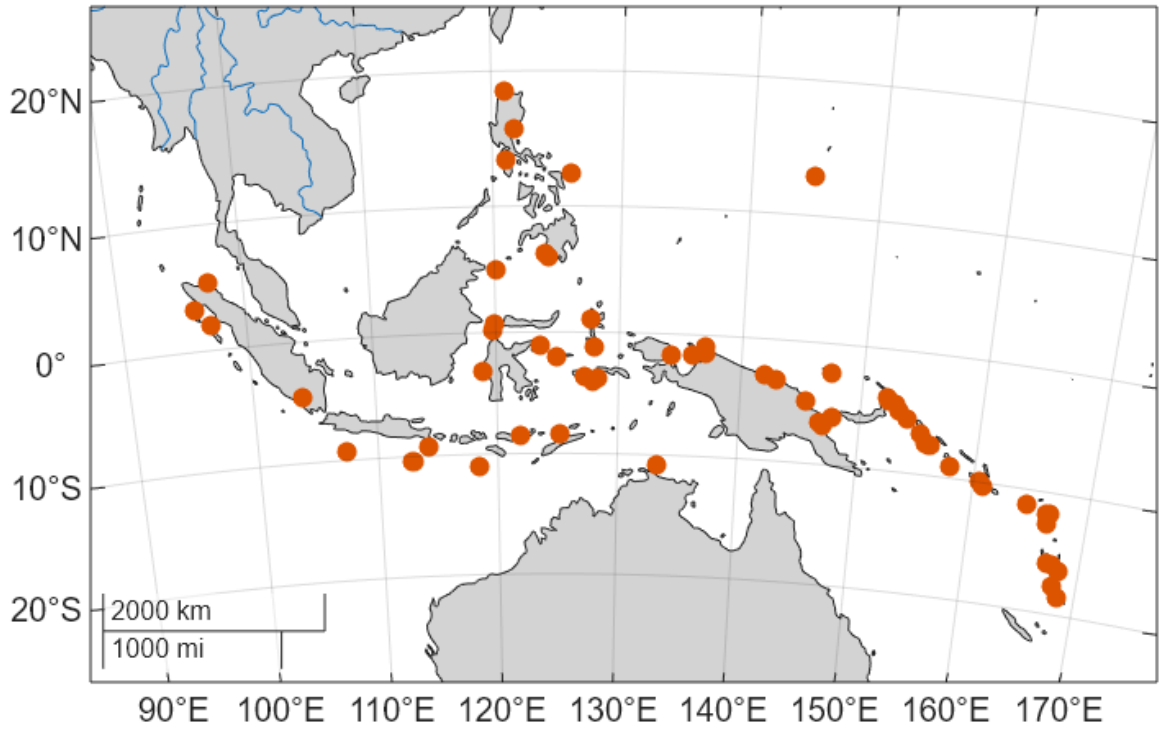
```
rivers = readgeotable("worldrivers.shp");
geoplot(rivers)
```

Display the locations of tsunamis events using points. Prepare to add data tips to the points by returning the `Point` object as `pt`.

```
tsunamis = readgeotable("tsunamis.shp",CoordinateSystemType="geographic");
pt = geoplot(tsunamis,MarkerSize=18);
```

Adjust the geographic limits.

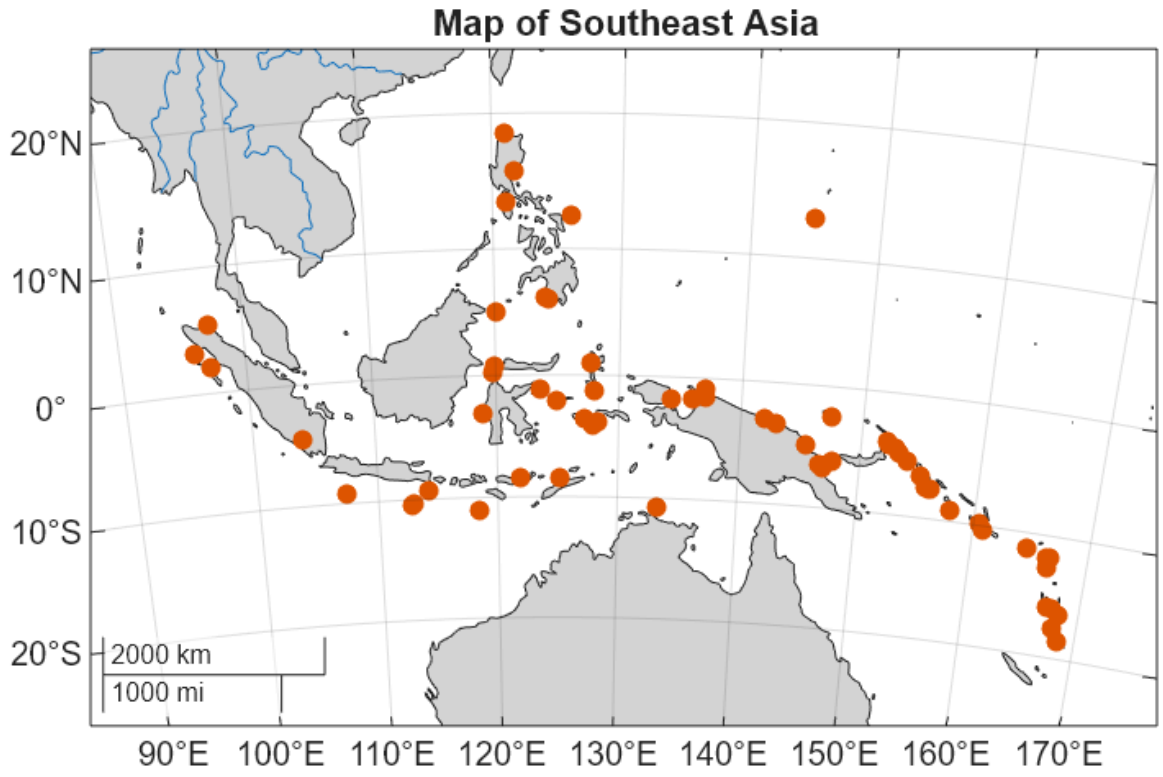
```
geolimits([-20 20],[90 170])
```



Add Title

Add a title using the `title` function.

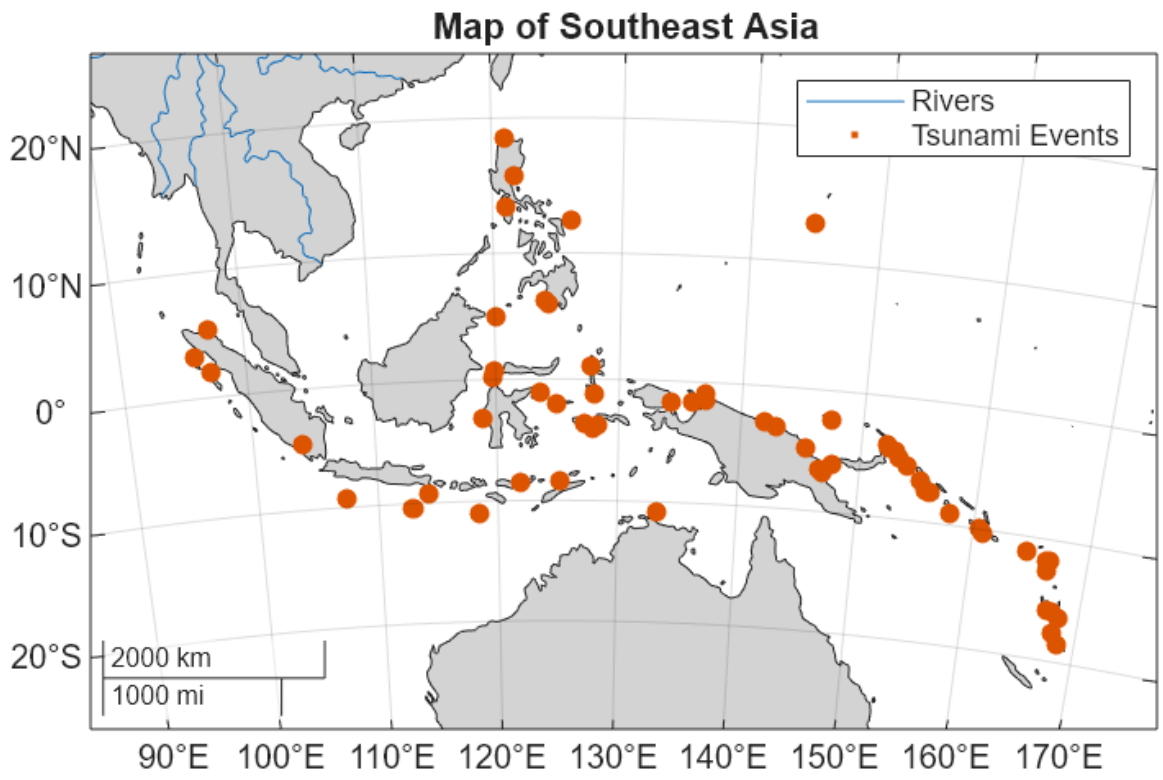
```
title("Map of Southeast Asia")
```



Add Legend

Add a legend with descriptions for the lines and scatter chart. Specify the legend labels as inputs to the `legend` function. For more information about adding legends to axes, see “Add Legend to Graph”.

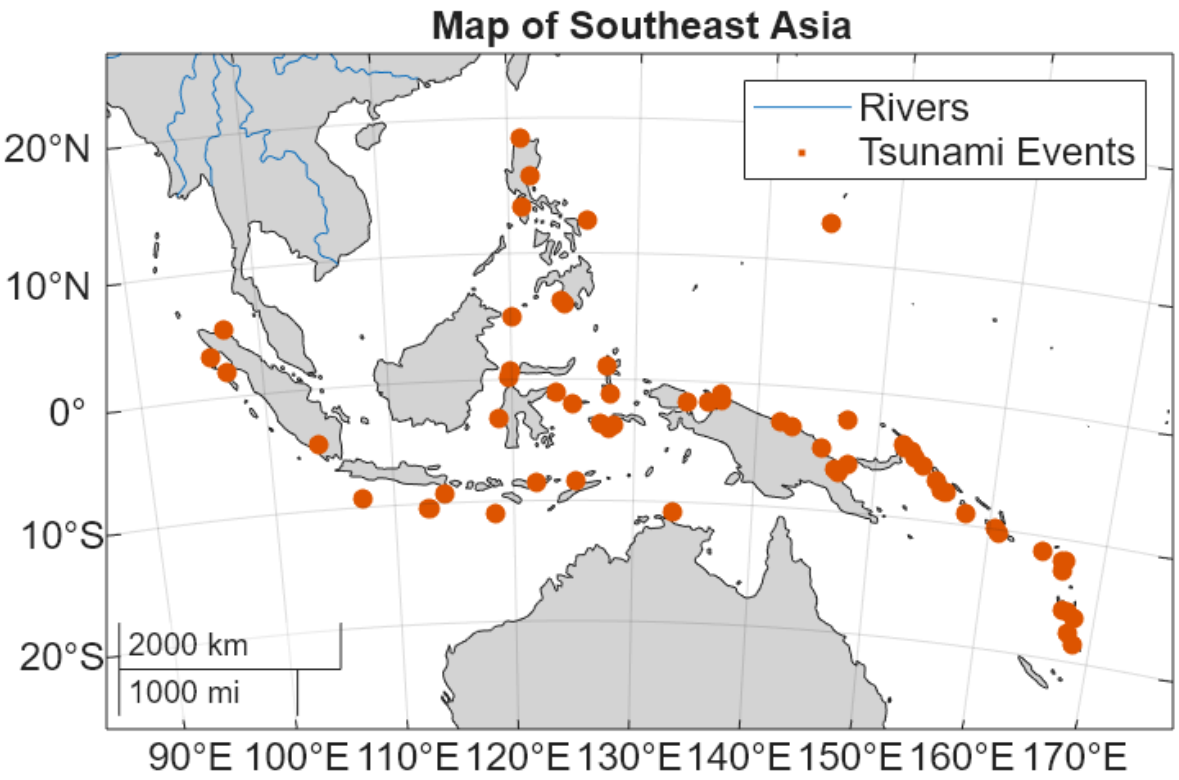
```
legend(["Rivers","Tsunami Events"],Location="northeast")
```



Change Font Size

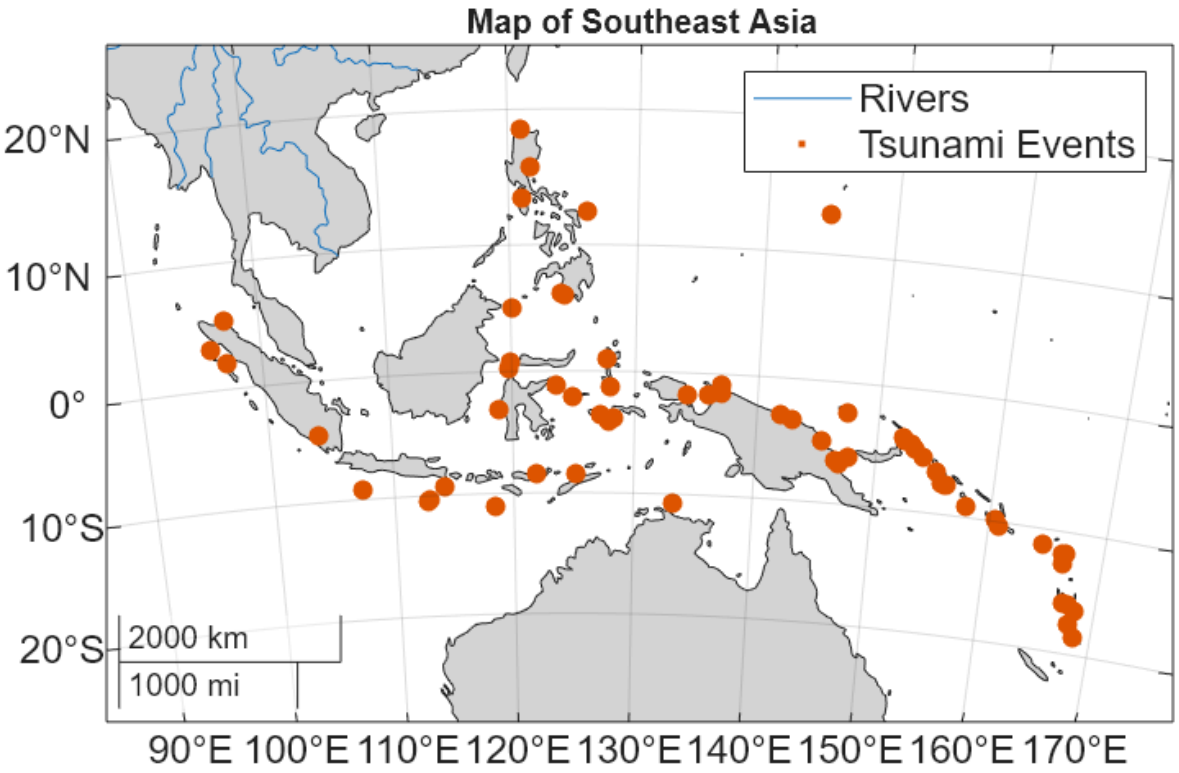
Change the font size by using the `fontsize` function. The `fontsize` function affects the sizes of fonts used in the title, tick labels, and scale bar, as well as any legends or color bars associated with the axes.

```
fontsize(12,"points")
```



To change the font size of only the title, access the title of the current `MapAxes` object. Then, set the `FontSize` property of the title.

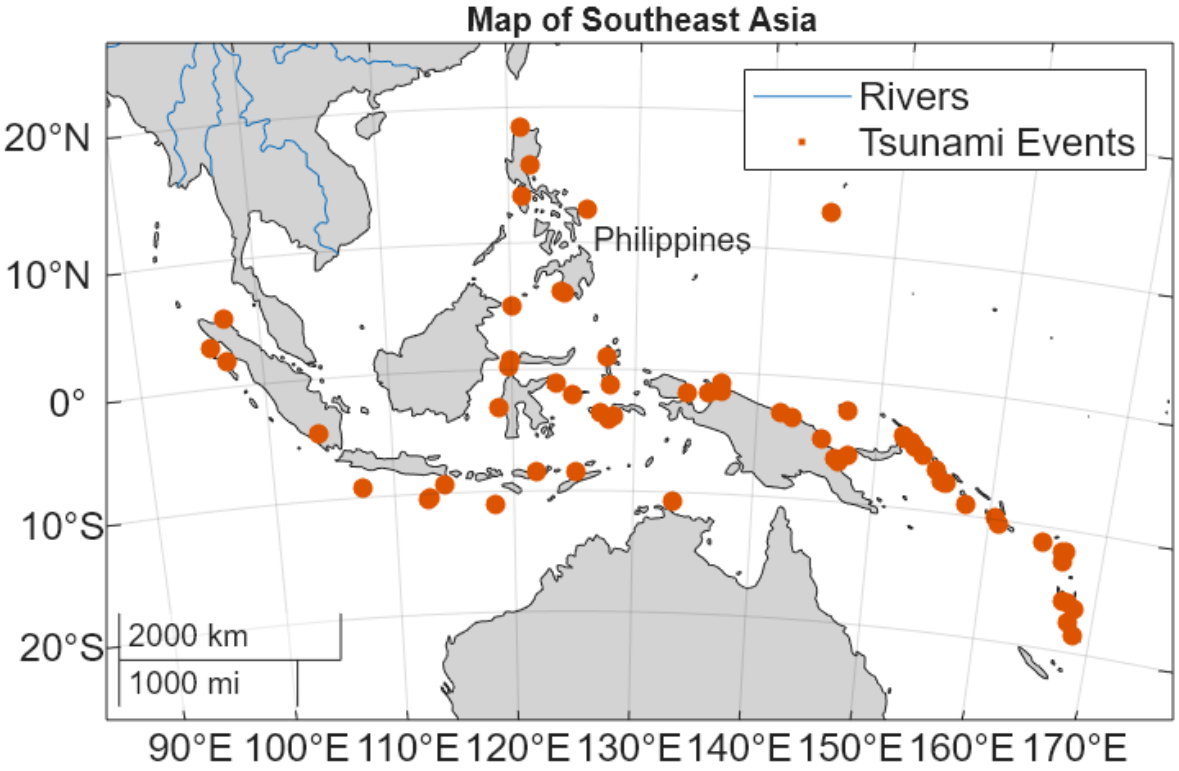
```
mx = gca;
mx.Title.FontSize = 10;
```



Add Text

Add text to the map. Specify the location for the text using latitude-longitude coordinates. For more information about adding text to axes, see “Add Text to Chart”.

```
text(10.2,126.5,"Philippines")
```



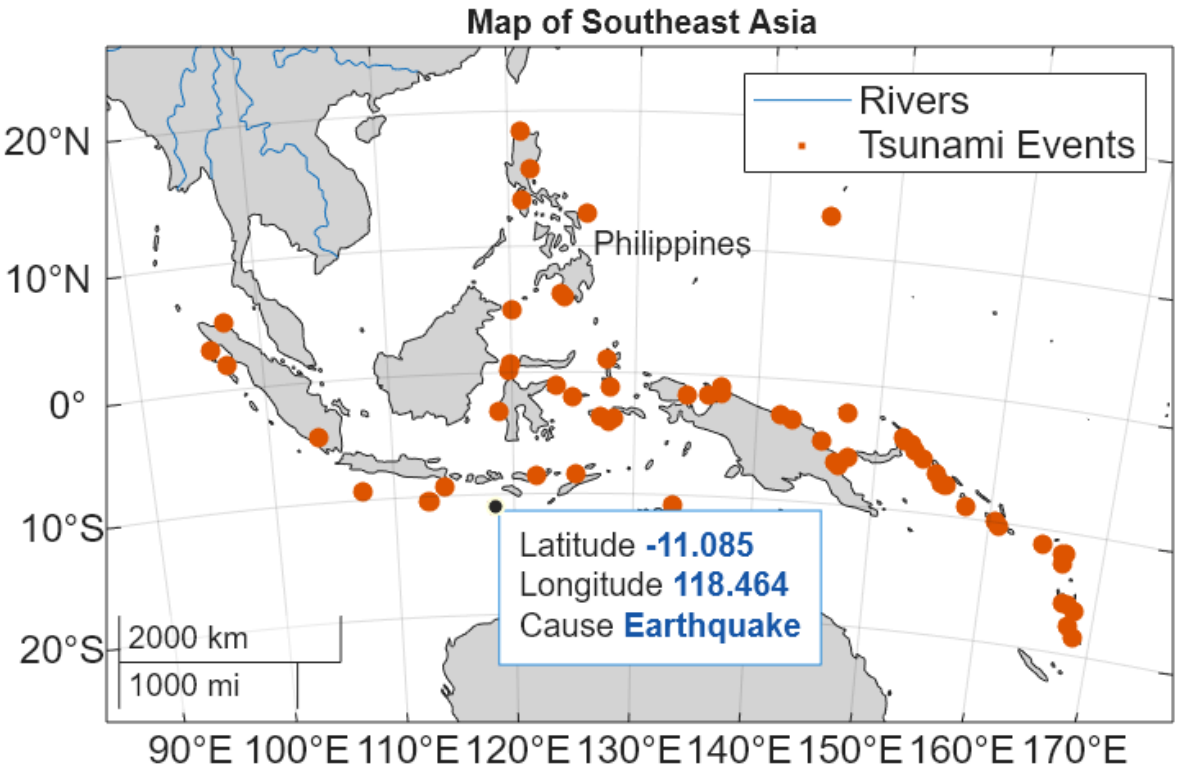
Add Data Tips

Display the coordinates of a tsunami event by adding a data tip. Specify the `Point` object and latitude-longitude coordinates as inputs to the `datatip` function.

```
dt = datatip(pt,-11.1,118.4,Location="southeast");
```

Display the cause of the tsunami event in the data tip by adding a new row to the data tip template. For more information about customizing data tips, see “Add Data Tips to Point, Line, and Polygon Shapes” on page 3-97.

```
dtRow = dataTipTextRow("Cause",tsunamis.Cause);
pt.DataTipTemplate.DataTipRows(end+1) = dtRow;
```



See Also

Functions
`title` | `legend` | `fontsize` | `text` | `datatip`

Properties
MapAxes Properties

Related Examples

- “Create Common Plots Using Map Axes” on page 3-6
- “Customize Appearance of Map Axes” on page 4-10

Customize Appearance of Map Axes

A map axes object is a type of axes object, similar to axes objects, geographic axes objects, and polar axes objects. This example shows how to customize the map axes by changing the tick label format, the geographic limits, the appearance of the graticule line, and the colors of fonts.

By default, most plotting functions reset many of the axes properties. As a result, this example customizes the properties of the axes after plotting the data. To customize map axes before plotting the data, set the hold state of the axes to `on` by using the `hold` function.

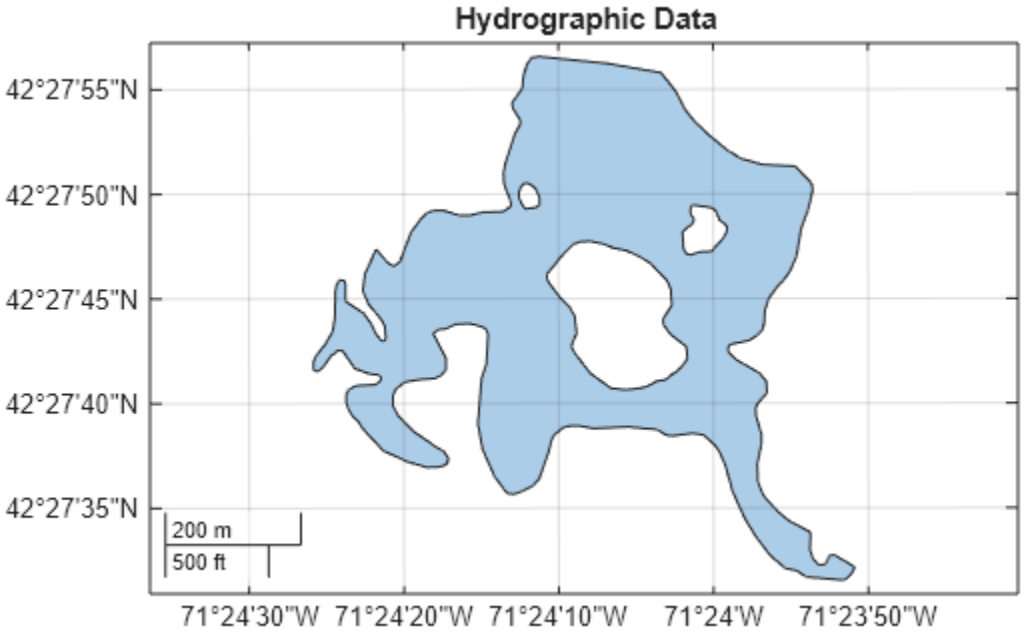
Create Map

Import a shapefile of hydrographic data into the workspace as a geospatial table. The table represents the data using polygon shapes in projected coordinates. Extract a row that contains data about a pond.

```
hydro = readgeotable("concord_hydro_area.shp");
pond = hydro(14,:);
```

Create a map using the projected coordinate reference system from the data file. Then, plot the pond polygon and add a title.

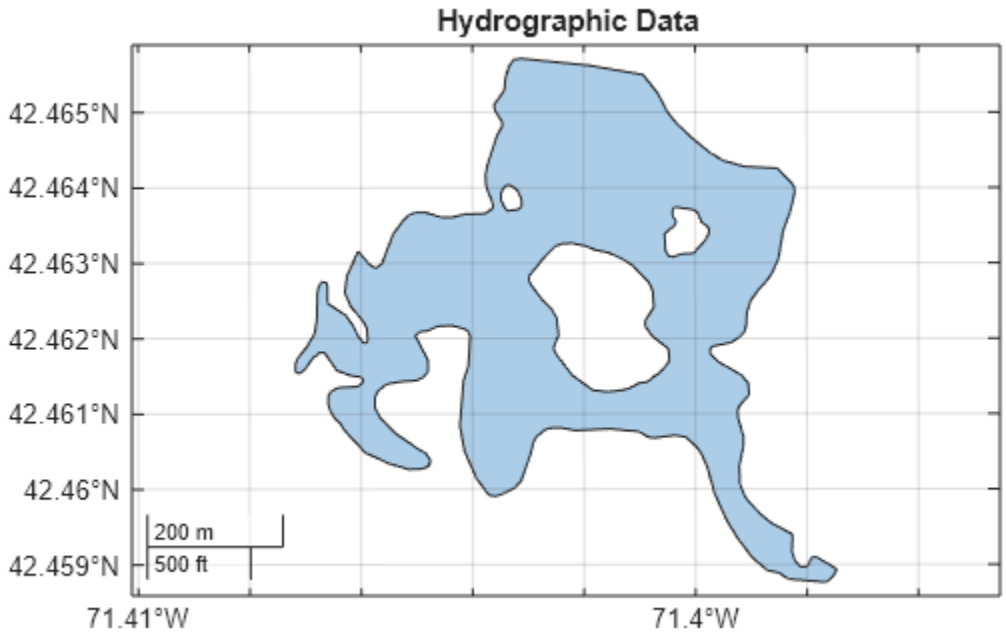
```
figure
newmap(pond.Shape.ProjectedCRS)

geoplot(pond)
title("Hydrographic Data")
```



Change Tick Format

Change the format of the tick labels to decimal degrees by using the `geotickformat` function.

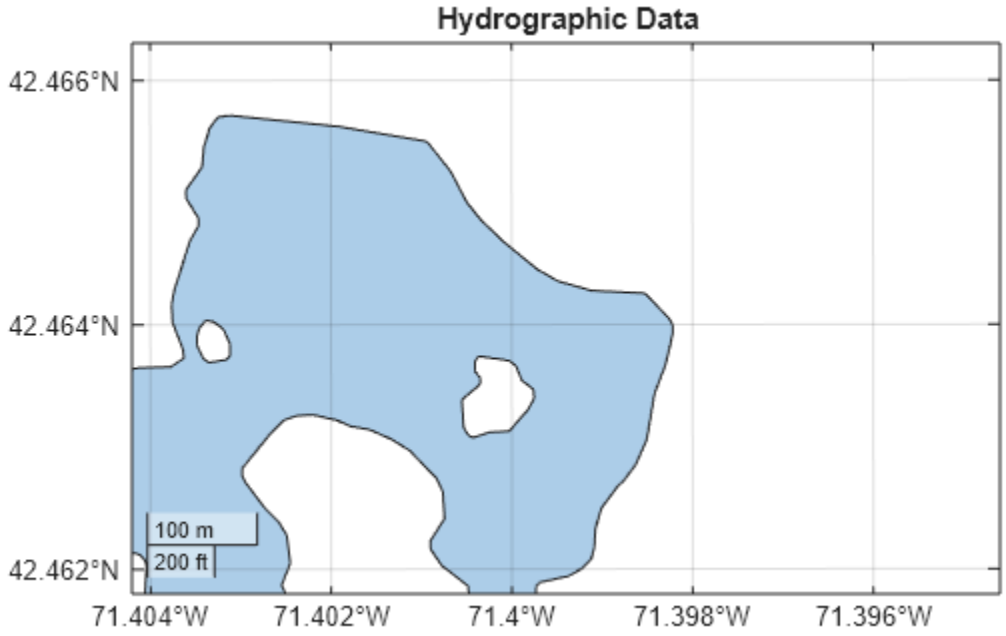
```
geotickformat dd
```



Change Geographic Limits

Change the latitude and longitude limits by using the `geolimits` function.

```
geolimits([42.4618 42.4663],[-71.4033 -71.3955])
```

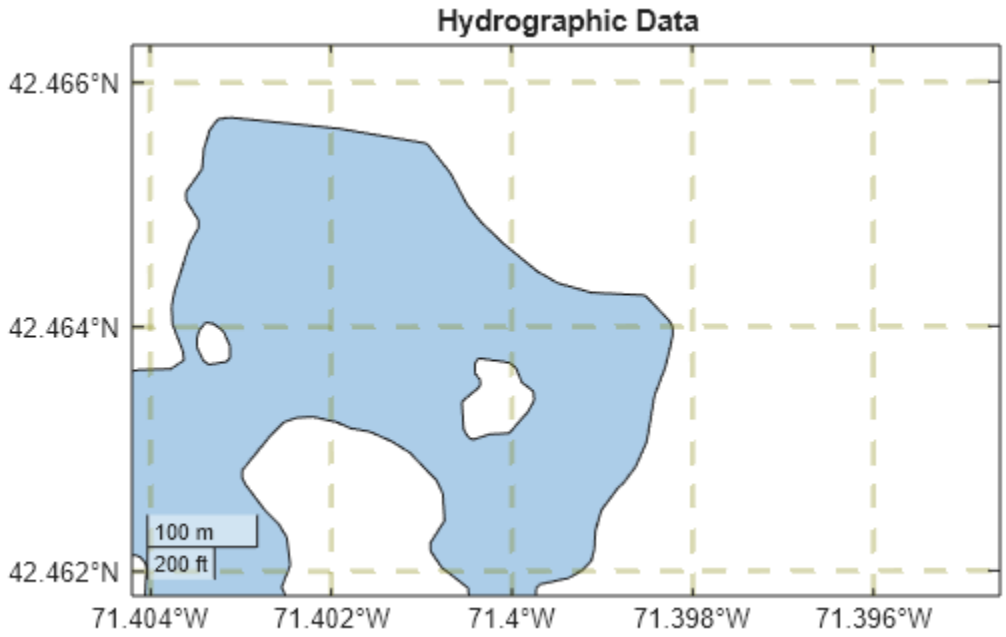


Change Appearance of Graticule Lines

`MapAxes` objects have properties that you can use to customize the appearance of the axes. For example, the `GraticuleColor`, `GraticuleLineStyle`, `GraticuleLineWidth`, and `GraticuleAlpha` properties control the color, line style, line width, and transparency of the graticule lines.

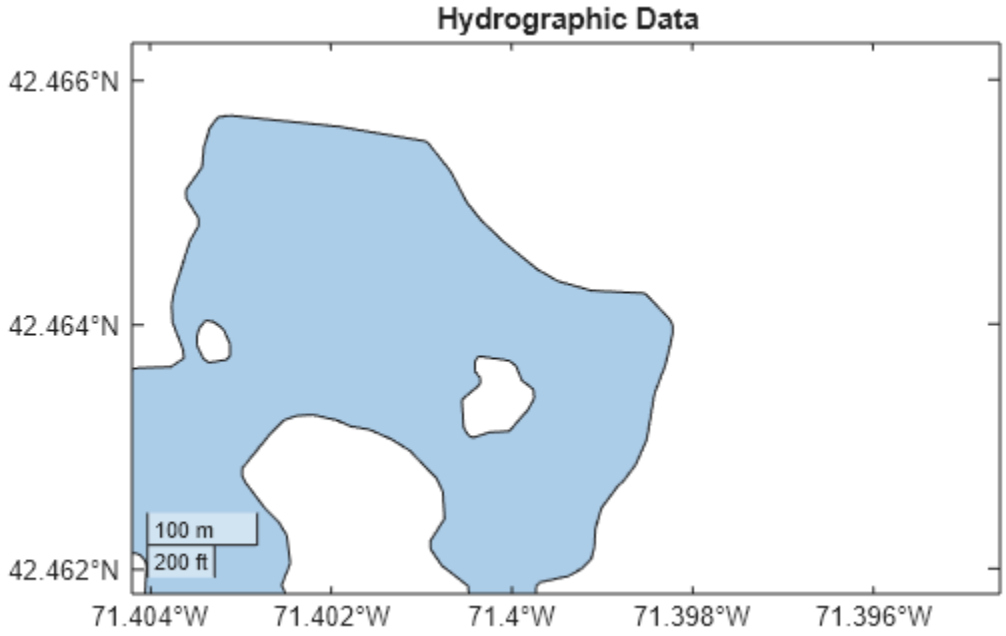
Access the `MapAxes` object and then change the properties of the graticule lines.

```
mx = gca;
mx.GraticuleColor = [0.5 0.5 0];
mx.GraticuleLineStyle = "--";
mx.GraticuleLineWidth = 2;
mx.GraticuleAlpha = 0.3;
```



Remove the graticule lines by setting the `GraticuleLineStyle` property to `"none"`.

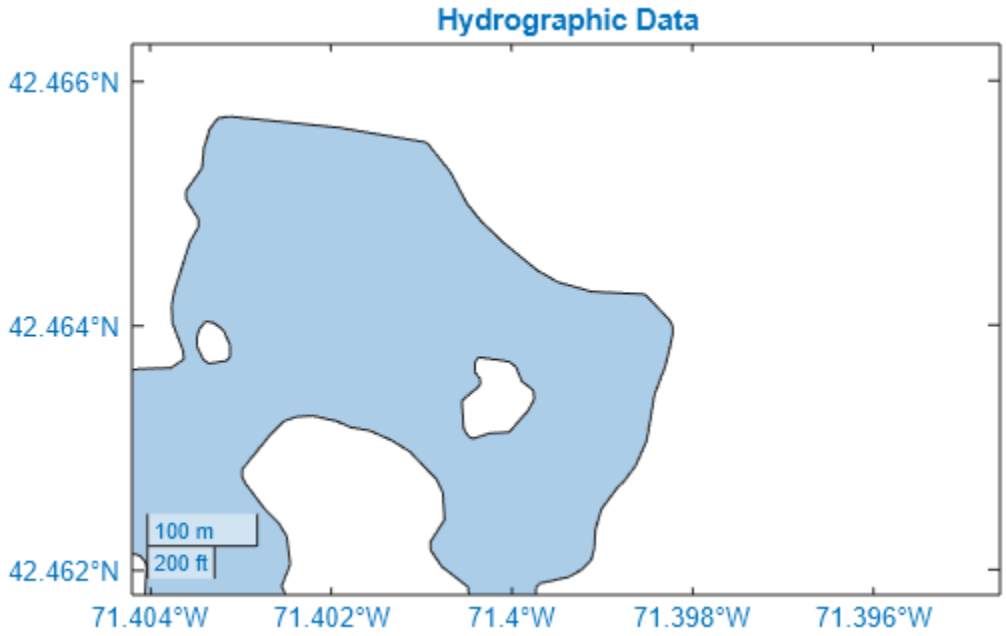
```
mx.GraticuleLineStyle = "none";
```



Change Font Color

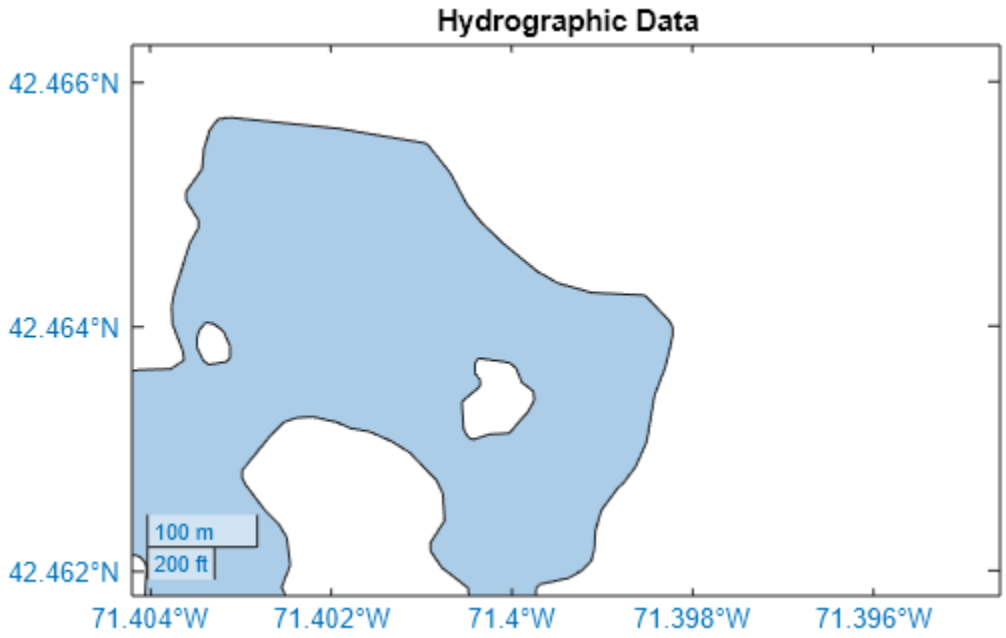
The `FontColor` property controls the font color for the title, tick labels, and scale bar. Change the font color of the map axes object.

```
mx.FontColor = [0 0.4470 0.7410];
```



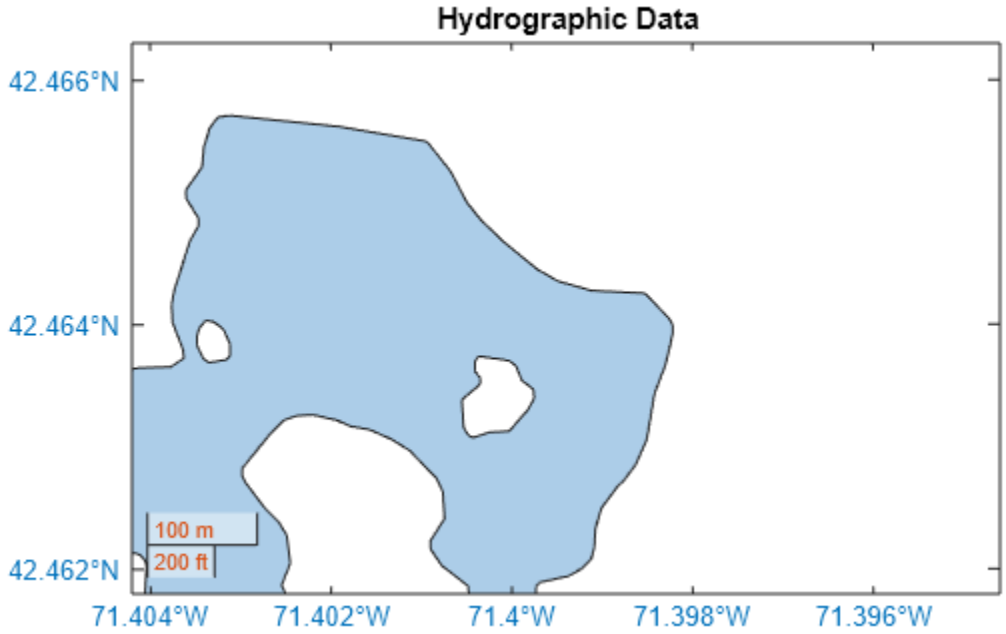
To change the color of only the title, set the `Color` property of the title.

```
mx.Title.Color = "k";
```



To change the font color for only the scale bar, set the `FontColor` property of the scale bar.

```
mx.Scalebar.FontColor = [0.8500 0.3250 0.0980];
```



See Also

Functions
`newmap` | `geoplot` | `geolimits` | `geotickformat`

Properties
MapAxes Properties

Related Examples

- “Create Common Plots Using Map Axes” on page 3-6
- “Add Labels and Annotations to Map Axes” on page 4-2

Create Map of Quadrangle Using Cartographic Map Layout

This example shows how to display data in map axes using a cartographic map layout. A cartographic map layout displays only the data within the quadrangle specified by the `CartographicLatitudeLimits` and `CartographicLongitudeLimits` properties of the map axes object. Cartographic map layouts are useful for creating static maps and for preparing maps for publications.

The default map layout is appropriate for most data visualization and exploration workflows. To change the geographic limits of a map in the default layout, use the `geolimits` function instead of the `CartographicLatitudeLimits` and `CartographicLongitudeLimits` properties.

By default, most plotting functions reset many of the axes properties. As a result, this example customizes the properties of the axes after plotting the data. To customize map axes before plotting the data, set the hold state of the axes to `on` by using the `hold` function.

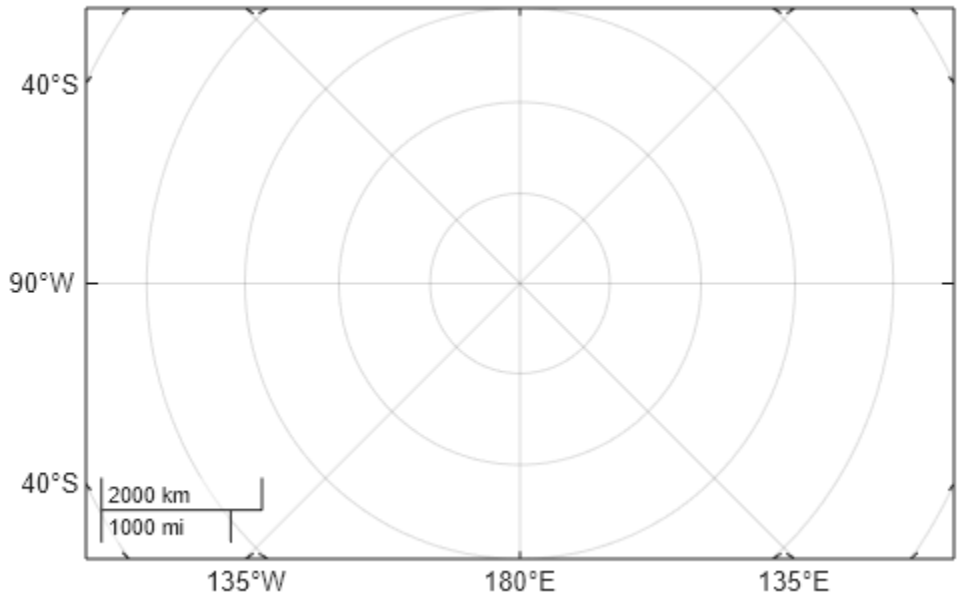
Use Default Cartographic Limits

Create a map in a cartographic map layout using the cartographic limits chosen by the map axes object.

Create Map Axes

Create a projected coordinate reference system (CRS) object that is appropriate for Antarctica. Use the WGS 84 / Antarctic Polar Stereographic projected CRS, which has the EPSG code `3031`. Then, create a map axes object that uses the projected CRS.

```
figure
p1 = projcrs(3031);
mx1 = newmap(p1);
```



By default, the map displays data in the box specified by the `Position` property of the axes object. For many projected CRSs, this box includes the quadrangle defined by the `CartographicLatitudeLimits` and `CartographicLongitudeLimits` properties and some areas surrounding the quadrangle. The axes object does not display data where the projection has undefined numeric results or extreme map distortion.

Read and Clip Data

Read a shapefile that contains world land areas into the workspace as a geospatial table. The table represents the land areas using polygon shapes in geographic coordinates.

```
land = readgeotable("landareas.shp");
```

Query the cartographic latitude and longitude limits stored in the map axes object. When you create a map axes object, the object sets the `CartographicLatitudeLimits` and `CartographicLongitudeLimits` properties using the area of use for the projected CRS.

```
latlim1 = mx1.CartographicLatitudeLimits
```

```
latlim1 = 1×2

   -90   -60
```

```
lonlim1 = mx1.CartographicLongitudeLimits
```

```
lonlim1 = 1×2

  -180   180
```

Clip the shapes to the cartographic latitude and longitude limits.

```
clipped1 = geoclip(land.Shape,latlim1,lonlim1);
```

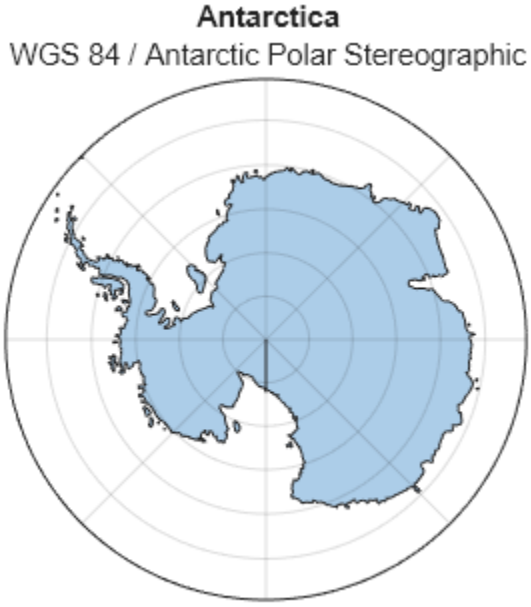
Display Data

Display the clipped shapes on the map axes object. Then, change the map layout to cartographic. By default, the map axes object hides the scale bar when the map outline is curved.

```
geoplot(clipped1)
mx1.MapLayout = "cartographic";
```

Add a title and subtitle.

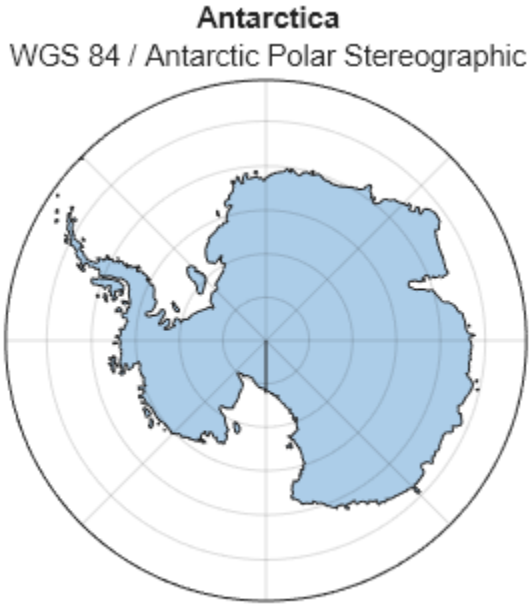
```
title("Antarctica")
subtitle(p1.Name)
```



Export Map

Export the map to a PNG file. Specify the resolution of the image as 600 dots per inch.

```
exportgraphics(mx1,"antarctica.png",Resolution=600)
```



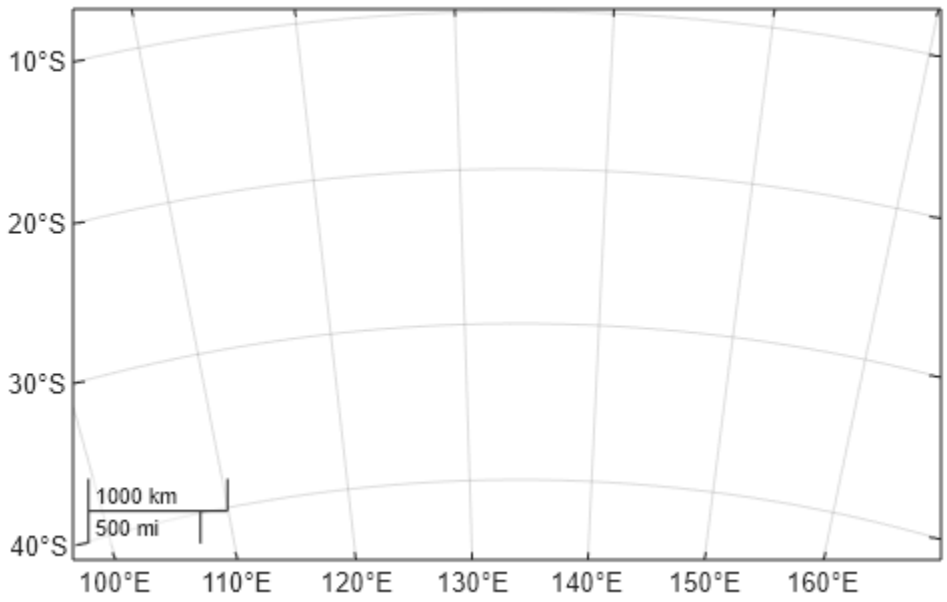
Change Cartographic Limits

Create a map in a cartographic map layout using cartographic limits that you specify. When you change the cartographic limits, the map axes object does not validate whether the limits are valid for the projected CRS. Use caution when changing the cartographic limit properties, for example, when specifying limits that are wider than the area of use for the projected CRS.

Create Map

Create a projected CRS object that is appropriate for Australia. Use the GDA94 / Geoscience Australia Lambert projected CRS, which has the EPSG code 3112. Then, create a map axes that uses the projected CRS.

```
figure
p2 = projcrs(3112);
mx2 = newmap(p2);
```



By default, the map displays data within the box specified by the `Position` property of the axes object. For many projected CRSs, this box includes the quadrangle defined by the `CartographicLatitudeLimits` and `CartographicLongitudeLimits` properties and some areas surrounding the quadrangle. The axes object does not display data where the projection has undefined numeric results or extreme map distortion.

Read and Clip Data

Read a shapefile that contains world land areas into the workspace as a geospatial table. The table represents the land areas using polygon shapes in geographic coordinates.

```
land = readgeotable("landareas.shp");
```

Query the cartographic latitude and longitude limits stored in the map axes object. When you create a map axes object, the object sets the `CartographicLatitudeLimits` and `CartographicLongitudeLimits` properties using the area of use for the projected CRS.

```
mx2.CartographicLatitudeLimits
```

```
ans = 1×2

  -43.7000   -9.8600
```

```
mx2.CartographicLongitudeLimits
```

```
ans = 1×2

  112.8500  153.6900
```

Specify latitude and longitude limits that are wider than the limits stored in the map axes object.

```
latlim2 = [-50 10];
lonlim2 = [88 180];
```

Clip the shapes to the wider limits.

```
clipped2 = geoclip(land.Shape,latlim2,lonlim2);
```

Display Data

Display the clipped shapes on the map axes object.

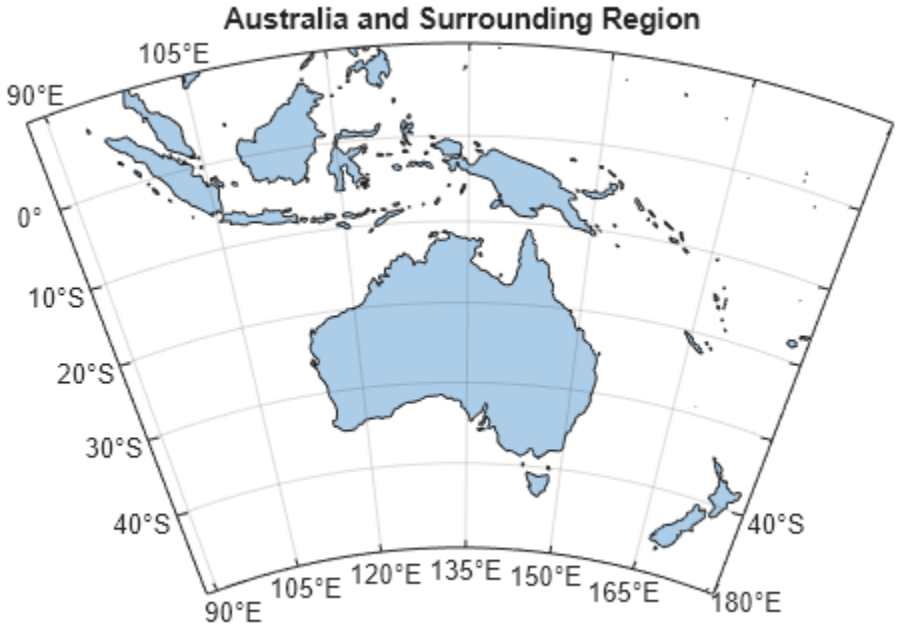
```
geoplot(clipped2)
```

Change the cartographic latitude and longitude limits to the wider limits. Then, change the map layout to cartographic. By default, the map axes object hides the scale bar when the map outline is curved.

```
mx2.CartographicLatitudeLimits = latlim2;
mx2.CartographicLongitudeLimits = lonlim2;
mx2.MapLayout = "cartographic";
```

Add a title and subtitle.

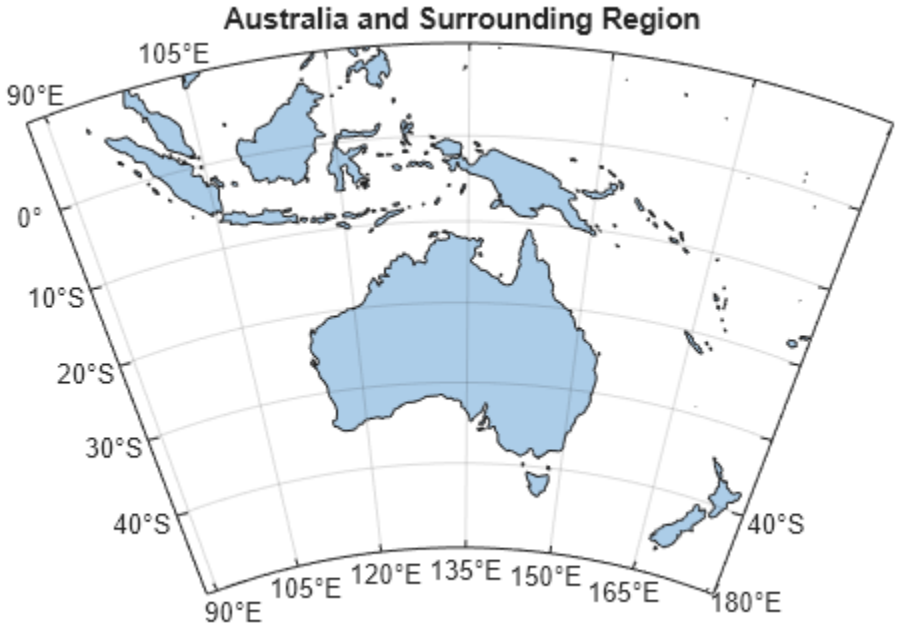
```
title("Australia and Surrounding Region")
```



Export Map

Export the map to a PNG file. Specify the resolution of the image as 600 dots per inch.

```
exportgraphics(mx2,"australia.png",Resolution=600)
```



Tips

Some projected CRSs do not indicate the area of use. When a projected CRS does not indicate the area of use, the map axes object sets the `CartographicLatitudeLimits` property to `[-90 90]` and set the `CartographicLongitudeLimits` property to `[-180 180]`.

See Also

Functions
`newmap` | `geoplot` | `exportgraphics`

Objects
`projcrs`

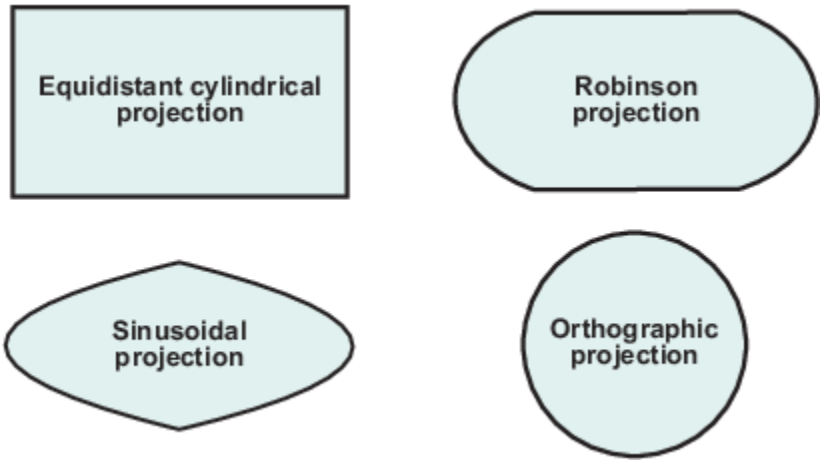
Properties
MapAxes Properties

Related Examples

- “Add Labels and Annotations to Map Axes” on page 4-2
- “Customize Appearance of Map Axes” on page 4-10

The Map Frame

The map frame of an `axesm`-based map is the outline of the limits of a map, often in the form of a *box*, the "edge of the world," so to speak. The frame is displayed if the `axesm`-based map property `Frame` is set to `'on'`. This can be accomplished upon `axesm`-based map creation with `axesm`, or later with `setm`, or with the direct command `framem on`. The frame is geographically defined as a latitude-longitude quadrangle that is projected appropriately. For example, on a map of the world, the frame might extend from pole to pole and a full 360° range of longitude. In appearance, the frame would take on the characteristic shape of the projection. The examples below are full-world frames shown in four very different projections.



Full-World Map Frames

As a map object, each of the previously displayed frames is identical; however, the selection of a display projection has varied their appearance.

You can manipulate properties beyond the latitude and longitude limits of the frame. Frame properties are established upon `axesm`-based map object creation; you can modify them subsequently with the `setm` and the `framem` functions. The command `framem` alone is a toggle for the `Frame` property, which controls the visibility of the frame. You can also call `framem` with property names and values to alter the appearance of the frame:

```
framem('FlineWidth',4,'FEdgeColor','red')
```

The frame is actually a patch with a default face color set to `'none'` and a default edge color of black. You can alter these `axesm`-based map properties by manipulating the `FFaceColor` and `FEdgeColor` properties. For example, the command

```
setm(gca,'FFaceColor','cyan')
```

makes the background region of your display resemble water. Since the frame patch is always the lowest layer of a map display, other patches, perhaps representing land, will appear above the "water." If an object is subsequently plotted "below" the frame patch, the frame altitude can be recalculated to lie below this object with the command `framem reset`. The frame is replaced and not reprojected.

Set the line width of the edge, which is 2 points by default, using the `FLineWidth` property.

The primary advantage of displaying the map frame is that it can provide positional context for other displayed map objects. For example, when vector data of the coasts is displayed, the frame provides the "edge" of the world.

See the `framem` reference page for more details.

Map and Frame Limits

The map and frame limits are two related `axesm`-based map properties that limit the map display to a defined region. The map latitude and longitude limits define the extents of geodata to be displayed, while the frame limits control how the frame fits around the displayed data. Any object that extends outside the frame limits is automatically trimmed.

The frame limits are also specified differently from the map limits. The map limits are in absolute geographic coordinates referenced to an origin at the intersection of the prime meridian and the equator, while the frame limits are referenced to the rotated coordinate system defined by the origin of the `axesm`-based map.

For all nonazimuthal projections, frame limits are specified as quadrangles (`[`*`latmin latmax`*`]` and `[`*`longmin longmax`*`]`) in the frame coordinate system. In the case of azimuthal projections, the frames are circular and are described by a polar coordinate system. One of the frame latitude limits must be a negative infinity (`-Inf`) to indicate an azimuthal frame (think of this as the center of the circle), while the other limit determines the radius of the circular frame (*`rlatmax`*). The longitude limits of azimuthal frames are inconsequential, since a full circle is always displayed.

If you are uncertain about the correct format for a particular projection frame limit, you can reset the formats to the default values using empty matrices.

Note For nonazimuthal projections in the normal aspect, the map extent is limited by the minimum of the map limits and the frame limits; hence, the two limits will coincide after evaluation. Therefore, if you manually change one set of limits, you might want to clear the other set to get consistent limits.

Map Limit Properties

In many common situations, the map limit properties of `axesm`-based maps, `MapLatLimit` and `MapLonLimit`, provide a convenient way of specifying your map projection origin or frame limits. Note that these properties are intentionally redundant; you can always avoid them if you wish and instead use the `Origin`, `FLatLimit`, and `FLonLimit` properties to set up your map. When they're applicable, however, you'll probably find that it's easier and more intuitive to set `MapLatLimit` and `MapLonLimit`, especially when creating a new `axesm`-based map.

You typically use the `MapLatLimit` and `MapLonLimit` properties to set up an `axesm`-based map with a non-oblique, non-azimuthal projection, with its origin on the Equator. (Most of the projections included in the Mapping Toolbox fall into this category; e.g., cylindrical, pseudo-cylindrical, conic, or modified azimuthal.) In addition, even with a non-zero origin latitude (origin off the Equator), you can use the `MapLatLimit` and `MapLonLimit` properties with projections that are implemented directly rather than via rotations of the sphere (e.g., `tranmerc`, `utm`, `lambertstd`, `cassinistd`, `eqaconicstd`, `eqdconicstd`, and `polyconicstd`). This list includes the projections used most frequently for large-scale maps, such as U.S. Geological Survey topographic quadrangle maps. Finally, when the origin is located at a pole or on the Equator, you can use the map limit properties with any azimuthal projection (e.g., `stereo`, `ortho`, `breusing`, `eqaazim`, `eqdazim`, `gnomonic`, or `vperspec`).

On the other hand, you should avoid the map limit properties, working instead with the `Origin`, `FLatLimit`, and `FLonLimit` properties, when:

- You want your map frame to be positioned asymmetrically with respect to the origin longitude.
- You want to use an oblique aspect (that is, assign a non-zero rotation angle to the third element of the "orientation vector" supplied as the `Origin` property value).
- You want to change your projection's default aspect (normal vs. transverse).
- You want to use a nonzero origin latitude, except in one of the special cases noted above.
- You are using one of the following projections:
  - `globe` — No need for map limits; always covers entire planet
  - `cassini` — Always in a transverse aspect
  - `wetch` — Always in a transverse aspect
  - `bries` — Always in an oblique aspect

There's no need to supply a value for the `MapLatLimit` property if you've already supplied one for the `Origin` and `FLatLimit` properties. In fact, if you supply all three when calling either `axesm` or `setm`, the `FLatLimit` value will be ignored. Likewise, if you supply values for `Origin`, `FLonLimit`, and `MapLonLimit`, the `FLonLimit` value will be ignored.

If you do supply a value for either `MapLatLimit` or `MapLonLimit` in one of the situations listed above, `axesm` or `setm` will ignore it and issue a warning. For example,

```
axesm('lambert','Origin',[40 0],'MapLatLimit',[20 70])
```

generates the warning message:

```
Ignoring value of MapLatLimit due to use of nonzero origin
 latitude with the lambert projection.
```

It's important to understand that `MapLatLimit` and `MapLonLimit` are extra, redundant properties that are coupled to the `Origin`, `FLatLimit`, and `FLonLimit` properties. On the other hand, it's not too difficult to know how to update your `axesm`-based map if you keep in mind the following:

- The `Origin` property takes precedence. It is set (implicitly, if not explicitly) every time you call `axesm` and you cannot change it just by changing the map limits. (Note that when creating a new `axesm`-based map, the map limits are used to help set the origin if it is not explicitly specified.)
- `MapLatLimit` takes precedence over `FLatLimit` if both are provided in the same call to `axesm` or `setm`, but changing either one alone affects the other.
- `MapLonLimit` and `FLonLimit` have a similar relationship.

The precedence of `Origin` means that if you want to reset your map limits with `setm` and have `setm` also determine a new origin, you must set `Origin` to [] in the same call. For example,

```
setm(gca,'Origin',[],'MapLatLimit',newMapLatlim,...
   'MapLonLimit',newMapLonlim)
```

On the other hand, a call like this will automatically update the values of `FLatLimit` and `FLonLimit`. Similarly, a call like:

```
setm(gca,'FLatLimit',newFrameLatlim,'FLonLimit',newFrameLonlim)
```

will update the values of `MapLatLimit` and `MapLonLimit`.

Finally, you probably don't want to try the following:

```
setm(gca,'Origin',[],'FLonLimit',newFrameLonlim)
```

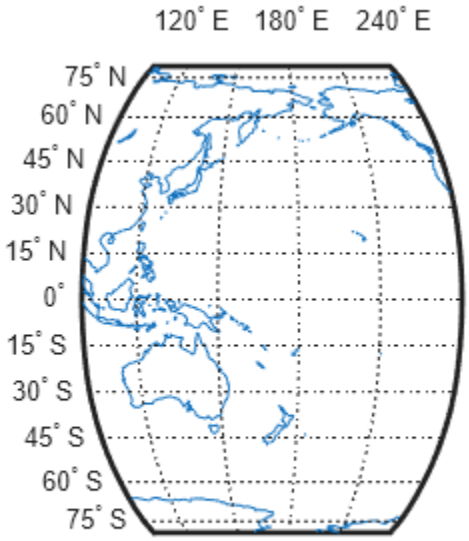
because the value of `FLonLimit` (unlike `MapLonLimit`) will not affect `Origin`, which will merely change to a projection-dependent default value (typically `[0 0 0]`).

Specify Map Projection Origin and Frame Limits Automatically

This example shows how to specify the map projection origin and frame limits of `axesm`-based maps using the two map limit properties: `MapLatLimit` and `MapLonLimit`. While `axesm`-based maps support properties to set these values directly, `Origin`, `FLatLimit`, and `FLonLimit`, it is easier and more intuitive to use the map limit properties, especially when creating a new `axesm`-based map. This example highlights the interdependency of the `axesm`-based map limits and the map limit properties.

Create a map using a cylindrical projection or pseudo-cylindrical projection showing all or most of the Earth, with the Equator running as a straight horizontal line across the center of the map. The map is bounded by a geographic quadrangle and the projection origin is located on the Equator, centered between the longitude limits you specify using the map projection limits.

```
latlim = [-80 80];
lonlim = [100 -120];
figure
axesm('robinson','MapLatLimit',latlim,'MapLonLimit',lonlim,...
    'Frame','on','Grid','on','MeridianLabel','on','ParallelLabel','on')
axis off
setm(gca,'MLabelLocation',60)
load coastlines
plotm(coastlat,coastlon)
```



Check that the `axesm` function set the origin and frame limits based on the values you specified using the `MapLatLim` and `MapLonLim` properties. The longitude of the origin should be located halfway between the longitude limits of 100 E and 120 W. Since the map spans 140 degrees, adding half of 140 to the western limit, the origin longitude should be 170 degrees. The frame is centered on this longitude with a half-width of 70 degrees and the origin latitude is on the Equator.

```
origin = getm(gca,'Origin')
```

```
origin = 1×3

     0   170     0
```

```
flatlim = getm(gca,'FLatLimit')
```

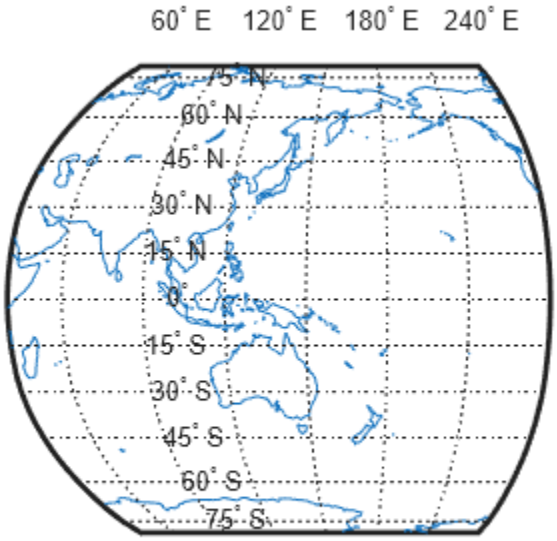
```
flatlim = 1×2

   -80    80
```

```
flonlim = getm(gca,'FLonLimit')
```

```
flonlim = 1×2

   -70    70
```

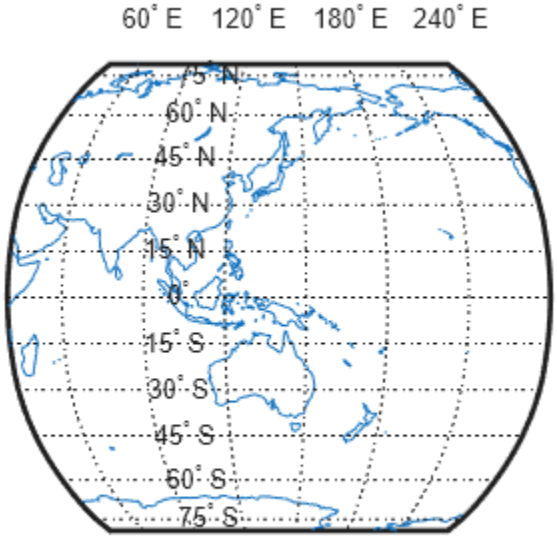
Shift the western longitude to 40 degrees E (rather than 100 degrees) to include a little more of Asia. Use the `setm` function to assign a new value to the `MapLonLimit` property. Note the asymmetric appearance of the map.

```
setm(gca,'MapLonLimit',[40 -120])
```



To correct the asymmetry, shift the western longitude again, this time specifying the origin. While the `MapLatLimit` and `MapLonLimit` properties are convenient, the values of the `Origin`, `FLatLimit`, and `FLonLimit` properties take precedence. You must specify the value of the origin to achieve the map you intended. The best way to do this is to specify an empty value for the `Origin` property and let the `setm` command calculate the value.

```
setm(gca,'MapLonLimit',[40 -120],'Origin',[])
```



Create Cylindrical Projection Using Map Limit Properties

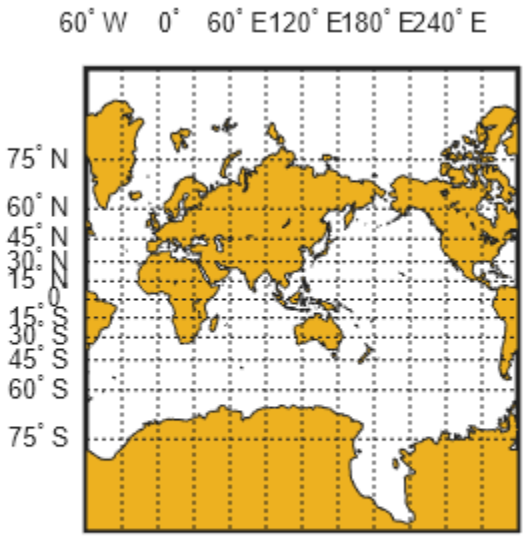
This example shows how to create cylindrical projection using map limit properties.

Load the coastline data.

```
load coastlines
```

Construct a Mercator projection covering the full range of permissible latitudes with longitudes covering a full 360 degrees starting at 60 West.

```
figure('Color','w')
axesm('mercator','MapLatLimit',[-90 90],'MapLonLimit',[-60 300])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
setm(gca,'MLabelLocation',60)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



The previous call to `axesm` is equivalent to:

```
axesm('mercator','Origin',[0 120 0],'FlatLimit',[-90 90],'FLonLimit',[-180
180]);
```

You can verify this by checking the properties.

```
getm(gca,'Origin')
```

```
ans = 1×3
```

```
     0   120     0
```

```
getm(gca,'FLatLimit')
```

```
ans = 1×2
```

```
   -86    86
```

```
getm(gca,'FLonLimit')
```

```
ans = 1×2
```

```
  -180   180
```

Note that the map and frame limits are clamped to the range of `[-86 86]` imposed by the read-only `TrimLat` property.

```
getm(gca,'MapLatLimit')
```

```
ans = 1×2
```

```
   -86    86
```

```
getm(gca,'FLatLimit')
```

```
ans = 1×2
```

```
   -86    86
```

```
getm(gca,'TrimLat')
```

```
ans = 1×2
```

```
   -86    86
```

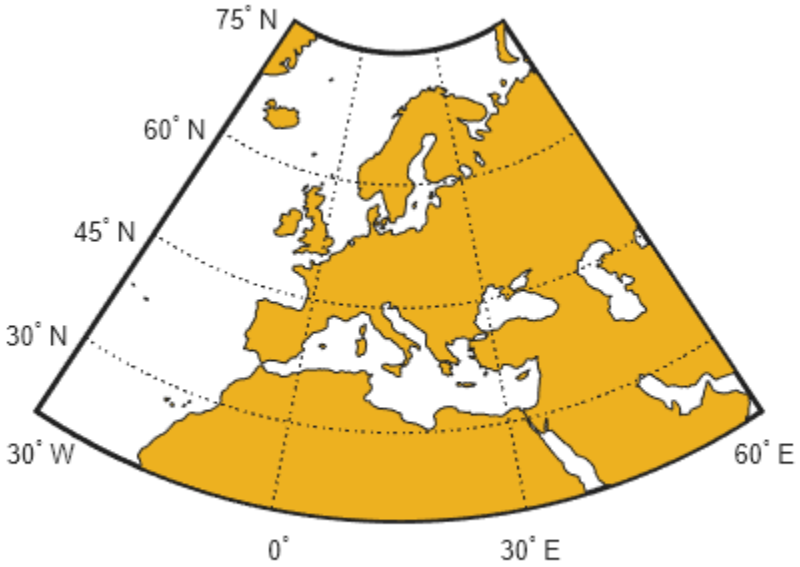
Create Conic Projection Using Map Limit Properties

This example shows how to create a map of the standard version of the Lambert Conformal Conic projection covering latitudes 20 North to 75 North and longitudes covering 90 degrees starting at 30 degrees West.

Load coastline data and display it. The call to `axesm` above is equivalent to: `axesm('lambertstd','Origin', [0 15 0], 'FLatLimit',[20 75],FLonLimit',[-45 45])`

```
load coastlines
figure('Color','w')
axesm('lambertstd','MapLatLimit',[20 75],'MapLonLimit',[-30 60])
axis off;
framem on;
```

```
gridm on;
mlabel on;
plabel on;
geoshow(coastlat, coastlon, 'DisplayType', 'polygon')
```



Create Southern Hemisphere Conic Projection

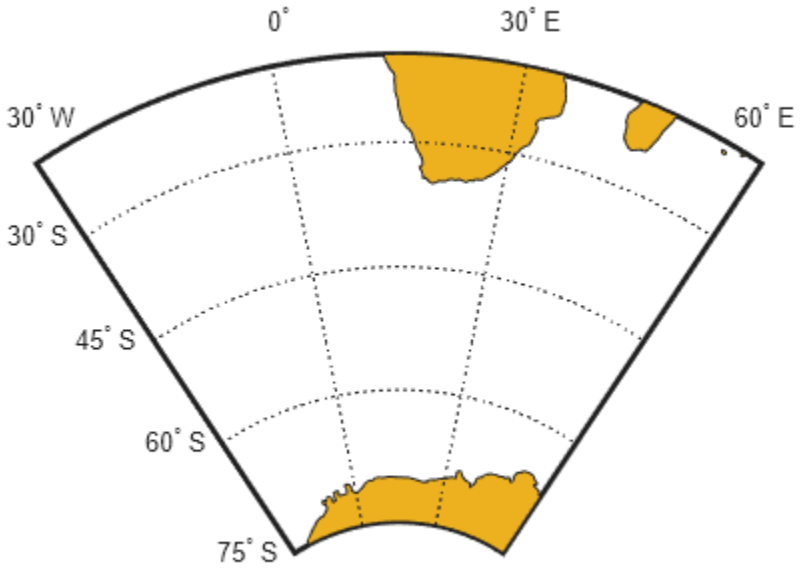
This example shows how to create a map of the standard version of the Lambert Conformal Conic projection into the Southern Hemisphere. The example overrides the default standard parallels and sets the `MapLatLimit` and `MapLonLimit` properties.

Load the coastline data MAT file, `coastlines.mat`.

```
load coastlines
```

Display the map, setting the `MapLatLimit` and `MapLonLimit` properties.

```
figure('Color','w')
axesm('lambertstd','MapParallels',[-75 -15], ...
        'MapLatLimit',[-75 -20],'MapLonLimit',[-30 60])
axis off
framem on
gridm on
mlabel on
plabel on
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



Create North-Polar Azimuthal Projection

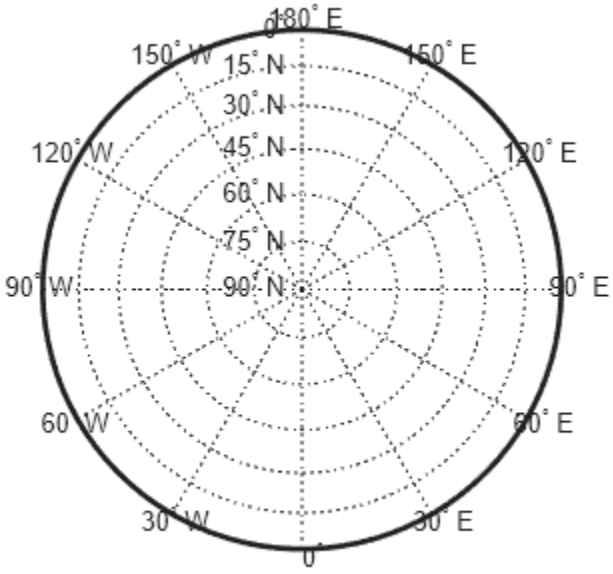
This example shows how to construct a North-polar Equal-Area Azimuthal projection map extending from the Equator to the pole and centered by default on longitude 0.

Load coastline data set MAT file, `coastlines.mat`.

```
load coastlines
```

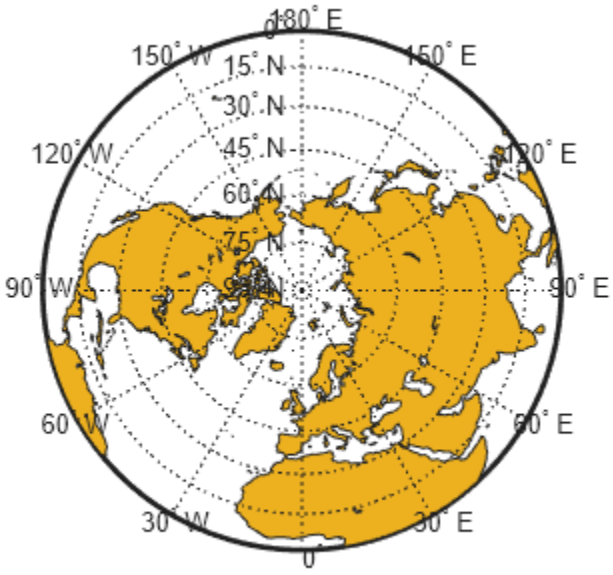
Create map. The call to `axesm` is equivalent to:
`axesm('eqaazim','MLabelParallel',0,'Origin',[90 0 0],'FLatLimit',[-Inf 90]);`

```
figure('Color','w')
axesm('eqaazim','MapLatLimit',[0 90])
axis off
framem on
gridm on
mlabel on
plabel on;
setm(gca,'MLabelParallel',0)
```



Plot the coast lines.

```
geoshow(coastlat,coastlon,'DisplayType','polygon')
```

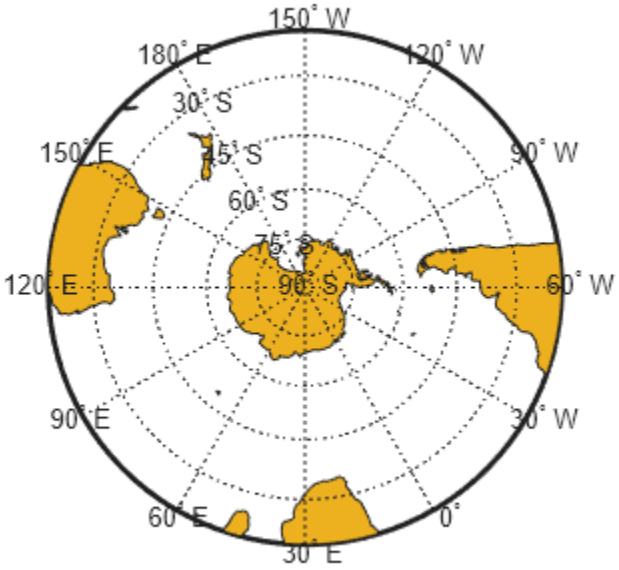


Create South-Polar Azimuthal Projection

This example shows how to create a South-polar Stereographic Azimuthal projection map extending from the South Pole to 20 degrees S, centered on longitude 150 degrees West. Include a value for the `Origin` property in order to control the central meridian.

Load coastline data and display map.

```
load coastlines
figure('Color','w')
axesm('stereo','Origin',[-90 -150],'MapLatLimit',[-90 -20])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
setm(gca,'MLabelParallel',-20)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



The call to the `axesm` function above is equivalent to:

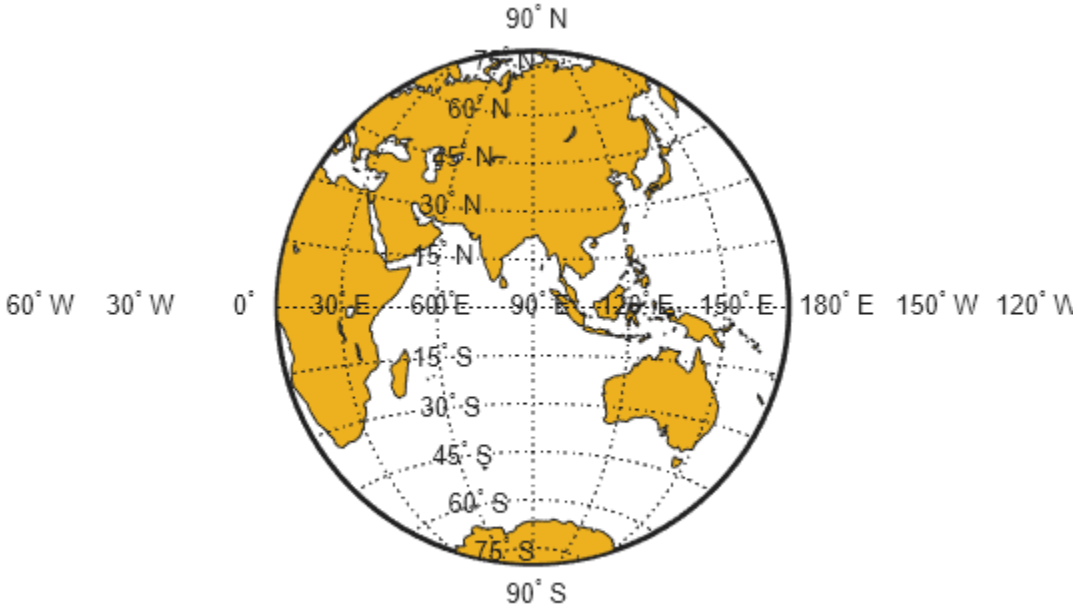
```
axesm('stereo','Origin',[-90 -150 0],'FLatLimit',[-Inf 70])
```

Create Equatorial Azimuthal Projection

This example shows how to create a map of an Equidistant Azimuthal projection with the origin on the Equator, covering from 10° E to 170° E. The origin longitude falls at the center of this range (90 E), and the map reaches north and south to within 10° of each pole.

Read coast data and display. The call to `axesm` is equivalent to `axesm('eqaazim','Origin',[0 90 0],'FLatLimit',[-Inf 80])`.

```
load coastlines
figure('Color','w')
axesm('eqdazim','FLatLimit',[],'MapLonLimit',[10 170])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
setm(gca,'MLabelParallel',0,'PLabelMeridian',60)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```

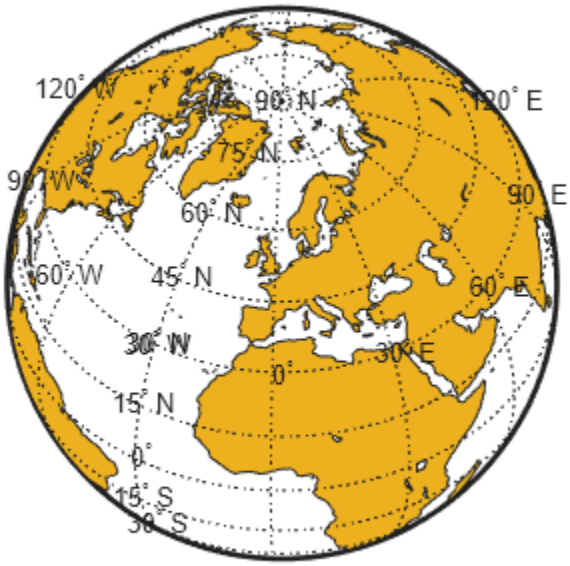


Create General Azimuthal Projection

This example shows how to construct an Orthographic projection map with the origin centered near Paris, France. You can't use `MapLatLimit` or `MapLonLimit` here.

Read in coast data and display.

```
load coastlines
originLat = dm2degrees([48 48]);
originLon = dm2degrees([ 2 20]);

figure('Color','w')
axesm('ortho','Origin',[originLat originLon])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelParallel',30,'PLabelMeridian',-30)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



Create Long Narrow Oblique Mercator Projection

This example shows how to create a map with a long, narrow, oblique Mercator projection. The example shows the area 10 degrees to either side of the great-circle flight path from Tokyo to New York. You can't use `MapLatLimit` or `MapLonLimit`.

```
load coastlines
latTokyo = dm2degrees([ 35 40]);
lonTokyo = dm2degrees([139 45]);

latNewYork = dm2degrees([ 40 47]);
lonNewYork = dm2degrees([-73 58]);

[dist,az] = distance(latTokyo,lonTokyo,latNewYork,lonNewYork);
[midLat,midLon] = reckon(latTokyo,lonTokyo,dist/2,az);
midAz = azimuth(midLat,midLon,latNewYork,lonNewYork);

buf = [-10 10];

figure('Color','w')
axesm('mercator','Origin',[midLat midLon 90-midAz], ...
    'FLatLimit',buf,'FLonLimit',[-dist/2 dist/2] + buf)
axis off; framem on; gridm on; tightmap
geoshow(coastlat,coastlon,'DisplayType','polygon')
plotm([latTokyo latNewYork],[lonTokyo lonNewYork],'r-')
```

See Also

More About

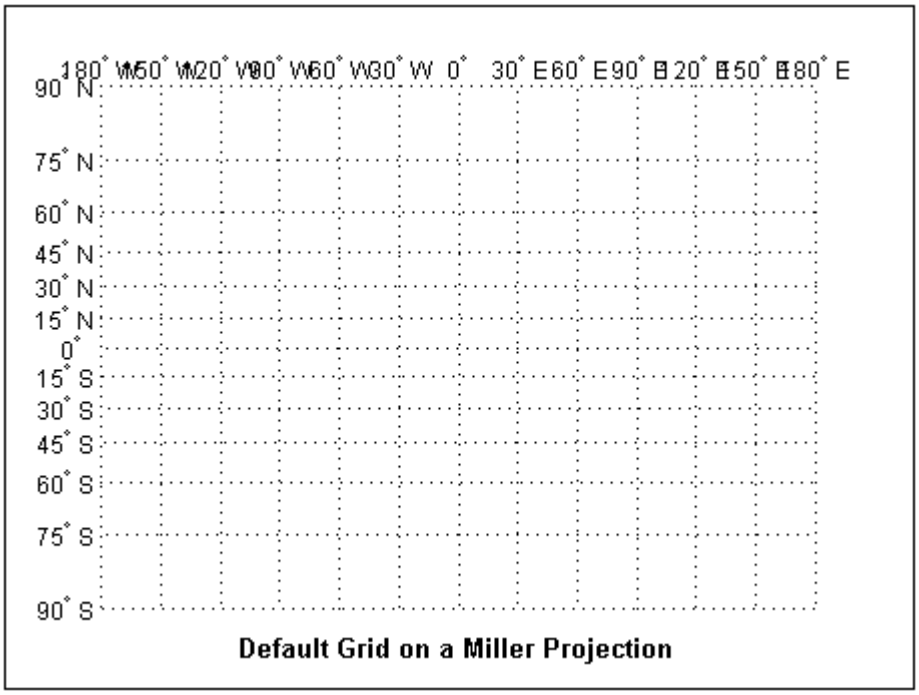
- “The Map Frame” on page 4-23
- “Map and Frame Limits” on page 4-25

The Map Grid

The *map grid* is the set of displayed meridians and parallels, also known as a *graticule*. Display the grid by setting the `axesm`-based map property `Grid` to `'on'`. You can do this when you create `axesm`-based maps with `axesm`, with `setm`, or with the direct command `gridm on`.

Control Grid Spacing

To control display of meridians and parallels, set a scalar meridian spacing or a vector of desired meridians in the `MLineLocation` property. The property `PLineLocation` serves a corresponding purpose for parallels. The default values place grid lines every 30° for meridians and every 15° for parallels.



Default Grid on a Miller Projection

Layer Grids

By default, the grid is placed as the top layer of any display. You can alter this by changing the `GAltitude` property, so that other map objects can be placed "above" the grid. The new grid is drawn at its new altitude. The units used for `GAltitude` are specified with the `daspectm` function.

To reposition the grid back to the top of the display, use the command `gridm reset`. You can also control the appearance of grid lines with the `GLineStyle` and `GLineWidth` properties, which are `':'` and `0.5`, respectively, by default.

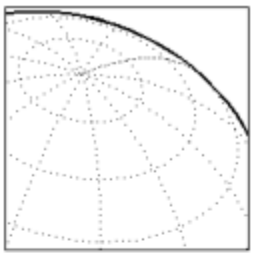
Limit Grid Lines

The Miller projection is an example in which all the meridians can extend to the poles without appearing to be cluttered. In other projections, such as the orthographic (below), the map grid can

obscure the surface where they converge. Two `axesm`-based map properties, `MLineLimit` and `MLineException`, enable you to control such clutter:

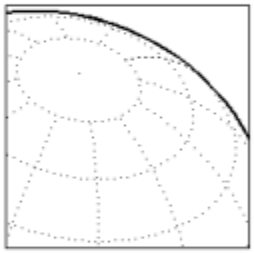
- Use the `MLineLimit` property to specify a pair of latitudes at which to terminate the meridians. For example, setting `MLineLimit` to `[-75 75]` completely clears the region above and below this latitude range of meridian lines.
- If you want some lines to reach the poles but not others, you can specify them with the `MLineException` property. For example, if `MLineException` is set to `[-90 0 90 180]`, then the meridians corresponding to the four cardinal longitudes will continue past the limit on to the pole.

The use of these properties is illustrated in the figure below. Note that there are two corresponding `axesm`-based map properties, `PLineLimit` and `PLineException`, for controlling the extent of displayed parallels.



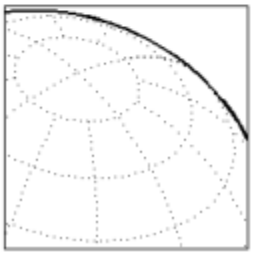
Default grid allows all displayed meridians to extend to the poles:

```
axesm('MapProjection','ortho',...
  'Origin',[40,40,14],...
  'Grid','on','Frame','on');
```



The property `MLineLimit` truncates meridians at given latitudes:

```
axesm('MapProjection','ortho',...
  'Origin',[40,40,14],...
  'Grid','on','Frame','on',...
  'MLineLimit', [-75 75]);
```



The property `MLineLineException` permits certain meridians to extend to the poles, regardless of `MLineLimit`:

```
axesm('MapProjection','ortho',...
  'Origin',[40,40,14],...
  'Grid','on','Frame','on',...
  'MLineLimit', [-75 75],...
  'MLineException',[-90 0 90 180]);
```

Label Grids

You can label displayed parallels and meridians. `MeridianLabel` and `ParallelLabel` are on-off properties for displaying labels on the meridians and parallels, respectively. They are both `'off'` by default. Initially, the label locations coincide with the default displayed grid lines, but you can alter this by using the `PlabelLocation` and `MlabelLocation` properties. These grid lines are labeled across the north edge of the map for meridians and along the west edge of the map for parallels. However, the property `MlabelParallel` allows you to specify `'north'`, `'south'`, `'equator'`, or a specific latitude at which to display the meridian labels, and `PlabelMeridian` allows the choice of `'west'`, `'east'`, `'prime'`, or a specific longitude for the parallel labels. By default, parallel labels are displayed in the range of 0° to 90° north and south of the equator while meridian labels are

displayed in the range of 0° to 180° east and west of the prime meridian. You can use the `mlabelzero22pi` function to redisplay the meridian labels in the range of 0° to 360° east of the prime meridian.

Properties affecting grid labeling are listed below.

Property	Effect
`MeridianLabel`	Toggle display of meridian labels
`ParallelLabel`	Toggle display of parallel labels
`MlabelLocation`	Alternate interval for labeling meridians
`PlabelLocation`	Alternate interval for labeling parallels
`MlabelParallel`	Keyword or latitude for placing meridian labels
`PlabelMeridian`	Keyword or longitude for placing parallel labels
`mlabelzero22pi`(function)	Relabel meridians with positive angle from 0° to 360°

For complete descriptions of all `axesm`-based map properties, refer to the axesm-Based Map Properties reference page.

Inset Maps

An inset map is a smaller map displayed inside a larger map. You can use an inset map to create geographic context for the larger map. This example shows how to create a map of Massachusetts and include an inset map of the northeastern United States.

This example shows how to create an inset map in an `axesm`-based map. For an example of creating inset maps in map axes, see “Position Multiple Map Axes in Figure” on the `mapaxes` reference page. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Prepare Data

Read a shapefile of US states into the workspace as a geospatial table. The table represents the states using polygon shapes in geographic coordinates.

```
states = readgeotable("usastatehi.shp");
```

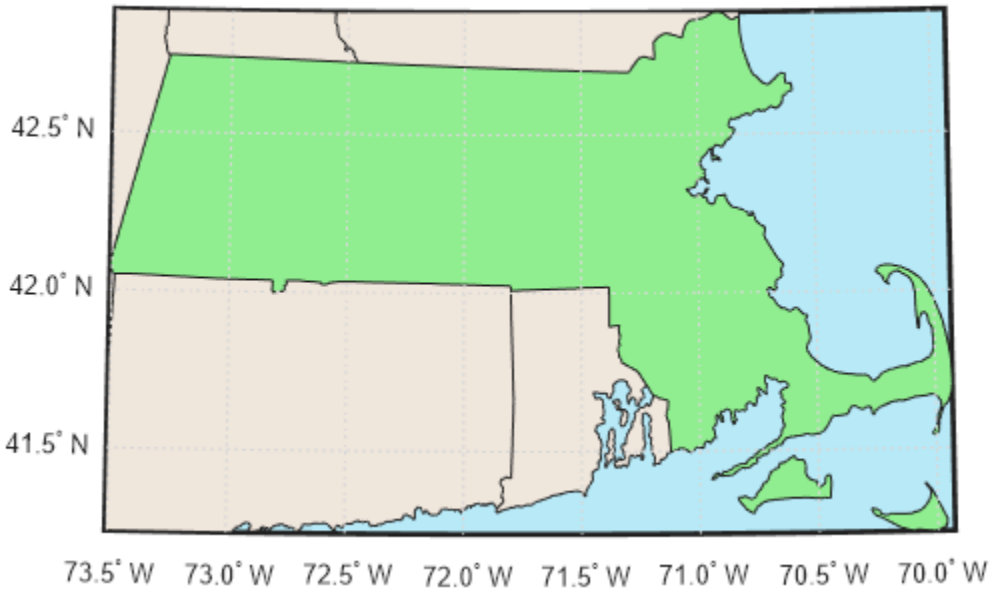
Extract the table row for Massachusetts. Find the latitude and longitude limits of the Massachusetts polygon shape.

```
ma = geocode("Massachusetts",states);
[latlim,lonlim] = bounds(ma.Shape);
```

Create Map

Create a map using the latitude and longitude coordinates. Set the face frame color of the map to light blue. Display the USA boundaries as light-brown polygons and the Massachusetts boundaries as green polygons.

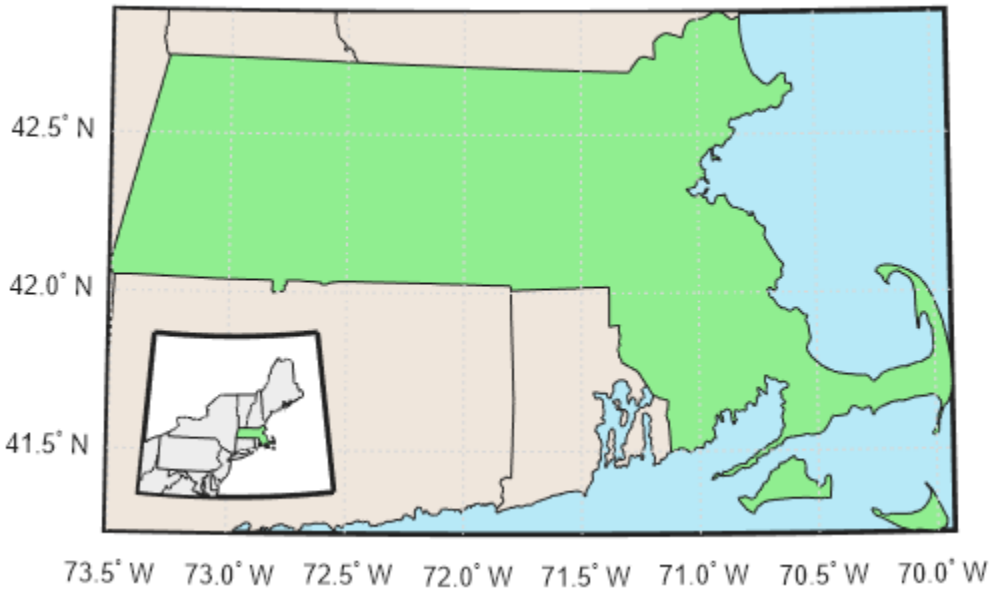
```
figure
h1 = usamap(latlim,lonlim);
setm(h1,"FFaceColor","#B7E9F7")
geoshow(states,FaceColor="#EFE6DC")
geoshow(ma,FaceColor="#90EE90")
```



Create Inset Map

Create an inset map. To do this, first create axes in the lower-left of the map frame. Place a map with latitude and longitude limits that contain Pennsylvania and Maine inside the axes, remove the parallel and meridian labels, and remove the grid lines. Then, display the USA boundaries as gray polygons and the Massachusetts boundaries as green polygons.

```
h2 = axes(Position=[0.13 0.18 0.25 0.25]);
usamap(["PA","ME"])
setm(h2,"FFaceColor","w")
plabel off
mlabel off
gridm off
geoshow(states,FaceColor="#EDEDED")
geoshow(ma,FaceColor="#90EE90")
```



See Also

Functions

`readgeotable` | `geocode` | `usamap` | `geoshow`

Related Examples

- “Create Map Displays with Geographic Data” on page 3-71

Customize Appearance of Vector Basemaps

Add custom vector basemaps from map tiles hosted by Esri® or from MBTiles files by using the `addCustomBasemap` function. You can customize the appearance of vector basemaps, such as the colors and fonts, by specifying a style. There are several ways to specify a style for vector basemaps.

- Use a predefined style on page 4-45 — Predefined styles are convenient and do not require internet access.
- Use a style URL on page 4-46 — Style URLs enable you to use visual elements that are not available with predefined styles.
- Use a JSON file on page 4-47 — JSON files are customizable and enable you to use visual elements that are not available with predefined styles.
- Edit a predefined style on page 4-48 — Editing a predefined style is available for vector basemaps added from MBTiles files.

Use a Predefined Style

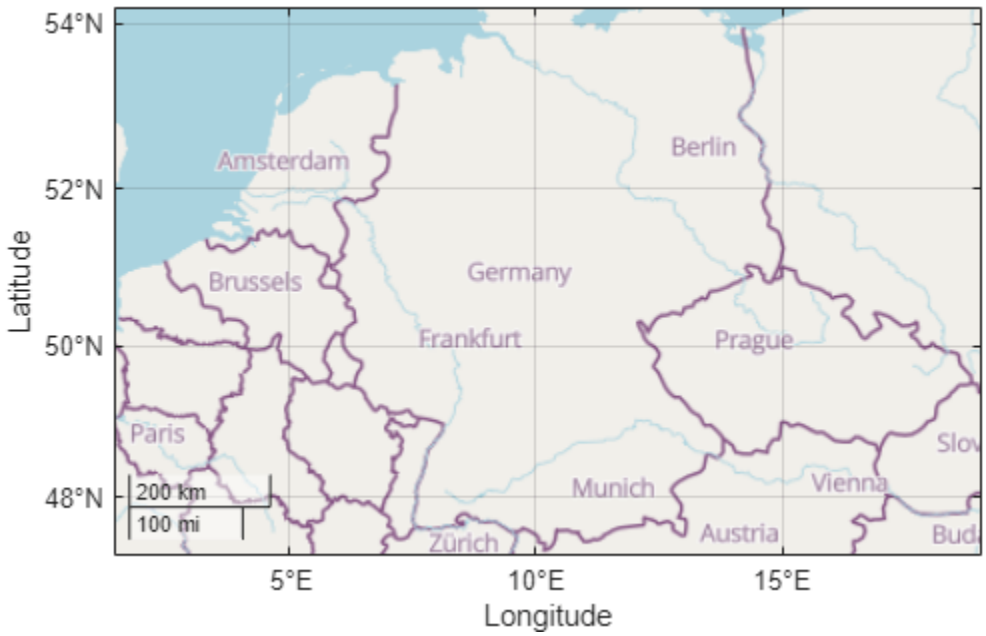
Specify the appearance of a custom vector basemap by using a predefined style. Predefined styles are convenient and do not require internet access. For a list of predefined style options, see the “Style” name-value argument on the `addCustomBasemap` page.

Create a basemap from an MBTiles file containing vector map tiles. Specify a predefined style that uses colors based on OpenStreetMap®.

```
basemapName1 = "osmstyle";
addCustomBasemap(basemapName1,"naturalearth.mbtiles",Style="openstreetmap")
```

Use the basemap to create a map of Germany.

```
figure
geobasemap(basemapName1)
geolimits([47.2 54.2],[1.50 19.01])
```



Use a URL

Specify the appearance of a custom vector basemap by using a URL. Styles defined by URLs enable you to use visual elements that are not available with predefined styles, for example, textures and additional fonts.

Create a basemap using the map tiles URL and style URL for Modern Antique [1] on page 4-50.

```
basemapName2 = "antiquestyle";
url2 = "https://basemaps.arcgis.com/arcgis/rest/services/" + ...
    "World_Basemap_v2/VectorTileServer/tile/${z}/${x}/${y}.pbf";
style2 = "https://www.arcgis.com/sharing/rest/content/items/" + ...
    "effe3475f05a4d608e66fd6eeb2113c0/resources/styles/root.json?f=pjson";
addCustomBasemap(basemapName2,url2,Style=style2)
```

Use the basemap to create a map of Germany.

```
figure
geobasemap(basemapName2)
geolimits([47.2 54.2],[1.50 19.01])
```

Use a JSON file

Specify the appearance of a custom vector basemap by using a JSON file. Styles defined by JSON files are customizable and enable you to use visual elements that are not available with predefined styles. For an example of how to edit JSON style files, see Edit a predefined style on page 4-48.

Create a basemap from an MBTiles file containing vector map tiles. Specify the style using a sample JSON file that emphasizes roads and transit networks.

```
basemapName3 = "streetstyle";
addCustomBasemap(basemapName3,"naturalearth.mbtiles",Style="streetstyle.json")
```

Use the basemap to create a map of Germany.

```
figure
geobasemap(basemapName3)
geolimits([47.2 54.2],[1.50 19.01])
```

Edit a Predefined Style

When you create a custom vector basemap using an MBTiles file and a predefined style, the `addCustomBasemap` function generates a JSON file that defines the style for the basemap. You can customize the appearance of a predefined style by editing the JSON file and then adding the basemap again. This example shows how to manually edit the JSON file using a text editor. Alternatively, if you have access to a visual style editor, you can customize the style interactively.

The `addCustomBasemap` function places the JSON file in the same folder as the MBTiles file. Prepare for the example by copying an MBTiles file containing vector map tiles to a writeable location, in this case, the working directory for the example.

```
mbtilesFilename = "naturalearth.mbtiles";
if ~isfile(mbtilesFilename)
    mbtilesFullPath = fullfile(matlabroot,"toolbox","map","mapdata",mbtilesFilename);
    copyfile(mbtilesFullPath)
end
```

Create a basemap from the MBTiles file. Specify a predefined style that uses light blue for water areas and light green for land areas.

```
basemapName4 = "bluegreenstyle";
addCustomBasemap(basemapName4,"naturalearth.mbtiles",Style="bluegreen")
```

Use the basemap to create a map of Germany.

```
figure
geobasemap(basemapName4)
geolimits([47.2 54.2],[1.50 19.01])
```

The `addCustomBasemap` function names the JSON file using the name of the MBTiles file and the name of the style, in this case, `naturalearth_bluegreen.json`. Open the file in a text editor. To programmatically open the file in the MATLAB® Editor, uncomment this code.

```
% edit("naturalearth_bluegreen.json")
```

The JSON file follows the Mapbox GL JS Style Specification [2] on page 4-50. Update the JSON file by making these changes:

- Background color — Find the layer that specifies the appearance of the background (`"id": "background"`). Change the background color to light orange by specifying the `"fill-color"` property as `"#f2de96"`.
- Land color — Find the layer that specifies the appearance of land areas (`"id": "land"`). Change the land color to light orange by specifying `"fill-color"` as `"#f2de96"`.
- Ocean color — Find the layer that specifies the appearance of ocean areas (`"id": "ocean"`). Change the ocean color to a different shade of blue by specifying `"fill-color"` as `"#add8e6"`.
- Country boundary width — Find the layer that specifies appearance of country boundaries [3] on page 4-50 (`"id": "admin0_boundary"`). The `"line-width"` property specifies line widths at different zoom levels. For example, at zoom level 4, the file specifies the line width as `1.5` pixels. Use thicker lines by specifying 2 pixels.
- Size of place markers — Find the layer that specifies the appearance of circle markers for high-importance places (`"id": "high_importance_places_circles"`). Change the radius of the markers by specifying the `"circle-radius"` property as 4.
- Size of place labels — Find the layer that specifies the appearance of text labels for populated places (`"id": "high_importance_places"`). The `"text-size"` property specifies the text size at different zoom levels. For example, at zoom level 2, the text size 11 pixels. Use larger text by specifying 13 pixels.

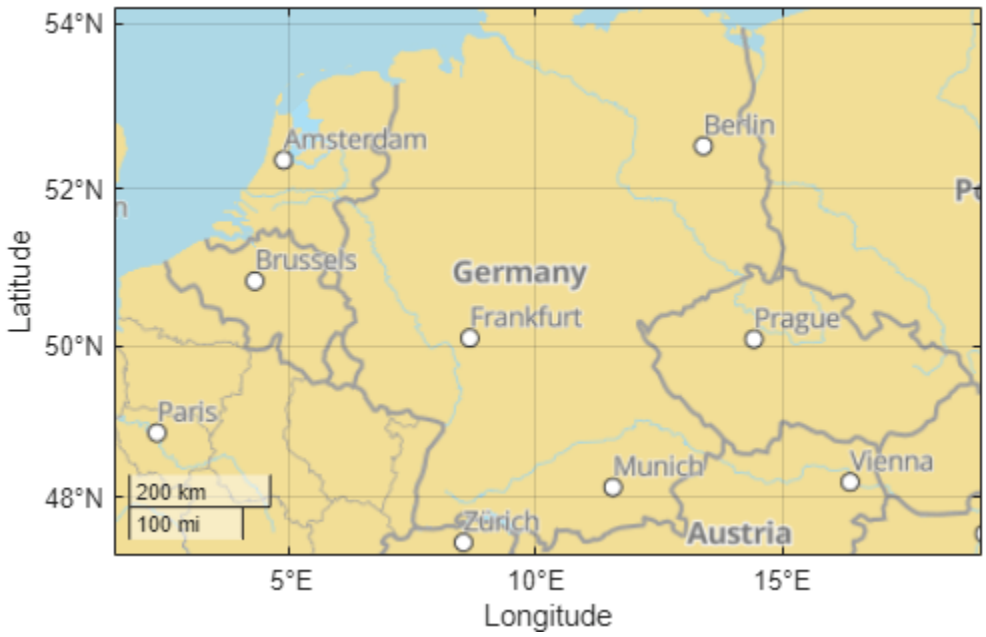
Save the updates to a new JSON file. This example includes the updates in the supporting file `naturalearth_blueorange.json`.

Add a custom basemap that uses the updated style by using the `addCustomBasemap` function.

```
basemapName5 = "blueorangestyle";
addCustomBasemap(basemapName5,mbtilesFilename,Style="naturalearth_blueorange.json")
```

View the updated basemap by creating a new map of Germany.

```
figure
geobasemap(basemapName5)
geolimits([47.2 54.2],[1.50 19.01])
```



[1] "Modern Antique - Overview." ArcGIS Online. Accessed December 8, 2022. https://www.arcgis.com/home/item.html?id=effe3475f05a4d608e66fd6eeb2113c0.

[2] "Style Specification | Mapbox GL JS | Mapbox." Mapbox. Accessed December 8, 2022. https://docs.mapbox.com/mapbox-gl-js/style-spec/.

[3] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks®.

See Also

Functions
`addCustomBasemap` | `geobasemap`

Related Examples

- “Use Basemaps in Offline Environments” on page 4-64

Create Common Plots over Basemap Images

When you create maps using geographic axes, you can provide context for your data by specifying a basemap. Geographic axes support several types of plots, including point, line, and polygon plots (`geoplot`), scatter plots (`geoscatter`), icon charts (`geoiconchart`), density plots (`geodensityplot`), and bubble plots (`geobubble` or `bubblechart`).

When geographic axes do not support a type of plot, you can read an image from a basemap by using the `readBasemapImage` function and then display both the image and the plot over an axes. This topic includes examples for displaying these types of plots over basemap images:

- Digraphs on page 4-52
- Contours on page 4-54
- Images on page 4-55
- 3-D surfaces on page 4-57

Digraphs

This example shows how to display a digraph of airport traffic over a basemap image.

Load Data

Specify the codes and geographic coordinates of five US airports.

```
code = ["DEN" "LAS" "LAX" "SEA" "SFO"];
lat = [39.8561 36.1699 33.9416 47.4502 37.6213];
lon = [-104.6737 -115.1398 -118.4085 -122.3088 -122.379];
```

Create a digraph from a table containing airport traffic data. Reorder the digraph nodes to match the order in `code`.

- The `Origin` and `Dest` table variables contain codes for the origin airports and destination airports.
- The `NumFlights` variable table contains the number of flights between the origin and destination airport.

```
T = readtable("airport_traffic.xlsx");
G = digraph(T.Origin,T.Dest,T.NumFlights);
G = reordernodes(G,code);
```

Read Basemap Image

Specify latitude and longitude limits for the basemap image. Create a quadrangular area of interest (AOI) from the airport locations. Buffer the AOI by 3 degrees. Then, find the latitude and longitude limits of the buffered AOI.

```
aoi = aoiquad(lat,lon);
aoi = buffer(aoi,3);
[latlim,lonlim] = bounds(aoi);
```

Read an image for the region from the `"landcover"` basemap, using a zoom level of `5`, as an array and a map cells reference object.

```
[A,RA] = readBasemapImage("landcover",latlim,lonlim,5);
```

Project Data to Web Mercator Coordinates

The `readBasemapImage` function spatially references basemap images by using a Web Mercator (WGS 84 / Pseudo-Mercator) projected coordinate reference system (CRS). Project the airport locations from geographic coordinates to Web Mercator coordinates.

```
[x,y] = projfwd(RA.ProjectedCRS,lat,lon);
```

Display Data over Basemap Image

Display the basemap image on a map with no axis labels.

```
figure
mapshow(A,RA)
hold on
axis off
```

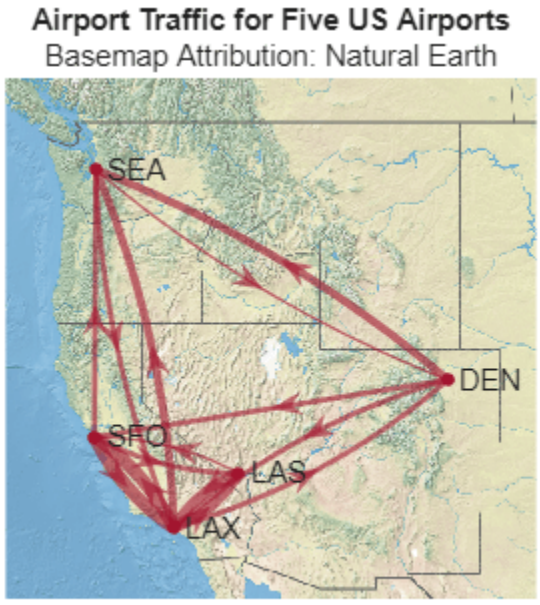
Display the digraph on the same map. Customize the digraph plot using these options:

- Position the nodes by setting the *x*-data and *y*-data properties to the projected airport coordinates.
- Use thicker edges for more heavily trafficked routes by setting the line width property.
- Improve the visibility of the plot by setting the node font size, edge transparency, arrow size, edge color, and node color properties.

```
lineWidth = G.Edges.Weight/min(G.Edges.Weight);
plot(G,XData=x,YData=y,LineWidth=lineWidth, ...
    NodeFontSize=11,EdgeAlpha=0.6,ArrowSize=12, ...
    EdgeColor="#A2142F",NodeColor="#A2142F")
```

Add a title and subtitle.

```
title("Airport Traffic for Five US Airports")
subtitle("Basemap Attribution: Natural Earth")
```



Contours

This example shows how to display contours of the Earth Gravitational Model of 1996 (EGM96) over a basemap image.

Load Data

Load geoid heights from EGM96 for a region containing the conterminous United States. The `egm96geoid` function returns geoid heights as an array and a geographic cells reference object in WGS 84 coordinates.

```
latlim = [17 56];
lonlim = [-128 -64];
[N,RN] = egm96geoid;
[N,RN] = geocrop(N,RN,latlim,lonlim);
```

To read spatially referenced raster data from a file, use the `readgeoraster` function.

Read Basemap Image

Read an image for the region from the `"satellite"` basemap as an array, a map cells reference object, and an attribution string.

```
[A,RA,attrib] = readBasemapImage("satellite",latlim,lonlim);
```

Project Data to Web Mercator Coordinates

The `readBasemapImage` function spatially references basemap images by using a Web Mercator (WGS 84 / Pseudo-Mercator) projected CRS. To display the geoid heights over the basemap image, you must project the geographic coordinates to the projected CRS.

Extract the geographic coordinates of the geoid heights from the reference object. Then, project the geographic coordinates to Web Mercator coordinates.

```
[lat,lon] = geographicGrid(RN);
[x,y] = projfwd(RA.ProjectedCRS,lat,lon);
```

Display Data over Basemap Image

Display the basemap image on a map with no axis labels.

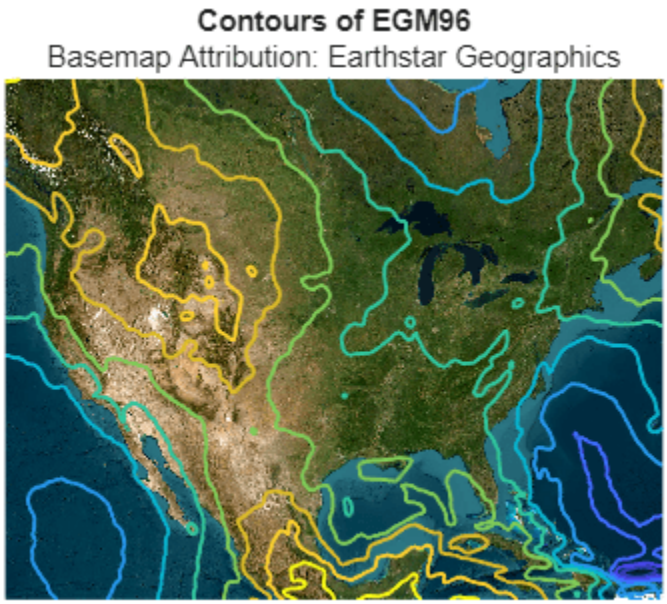
```
figure
mapshow(A,RA)
hold on
axis off
```

Display `10` contours of the geoid heights on the same map.

```
contour(x,y,N,10,LineWidth=1.5)
```

Add a title and subtitle.

```
title("Contours of EGM96")
subtitle("Basemap Attribution: " + attrib)
```



Images

This example shows how to display a photograph taken on Berkeley Street in Boston, MA over a basemap image.

Read Basemap Image

Specify the latitude and longitude coordinates of the camera. Then, read an image of size `800`-by-`1000` from the `"streets"` basemap for a region centered on the camera. The `readBasemapImage` function returns the basemap image as an array, a map cells reference object, and an attribution string.

```
lat = 42.35009;
lon = -71.07244;
[A,RA,attrib] = readBasemapImage("streets",[lat lon],18,[800 1000]);
```

Load Photograph

Read the photograph as an RGB image of size `480`-by-`640`-by-`3` and a colormap. To ensure the photograph fits within the basemap image, decrease the size of the photograph.

```
P = imread(fullfile(matlabroot,"toolbox","matlab","demos","street1.jpg"));
P = imresize(P,0.6);
```

Spatially Reference Photograph

The `readBasemapImage` function spatially references basemap images by using a Web Mercator (WGS 84 / Pseudo-Mercator) projected CRS. Using the coordinates of the camera as the lower-left corner of the image, spatially reference the photograph to Web Mercator coordinates.

Project the latitude and longitude coordinates of the camera to Web Mercator coordinates.

```
[x,y] = projfwd(RA.ProjectedCRS,lat,lon);
```

Specify the *xy*-limits of the photograph using the projected coordinates, the size of the photograph, and the sizes of individual cells in the basemap image.

```
sz = size(P);
xlimits = [x (x + sz(2)*RA.CellExtentInWorldX)];
ylimits = [y (y + sz(1)*RA.CellExtentInWorldY)];
```

Spatially reference the photograph by creating a map cells reference object. Indicate that the first row of the photograph is the northmost row (as is common with images) by setting the `ColumnsStartFrom` property.

```
RP = maprefcells(xlimits,ylimits,sz,ColumnsStartFrom="north");
```

Display Data over Basemap Image

Display the basemap image on a map with no axis labels.

```
figure
mapshow(A,RA)
hold on
axis off
```

Display the photograph and the location of the camera on the same map.

```
mapshow(P,RP)
mapshow(x,y,DisplayType="point",MarkerSize=7,Marker="o",MarkerFaceColor="r")
```

Add a title and subtitle. Wrap the basemap attribution text.

```
title("Image Over Basemap")

t = "Basemap Attribution: " + attrib;
t = string(textwrap(t,70));
subtitle(t,FontSize=9)
```

3-D Surfaces

This example shows how to display a sample 3-D surface over a basemap image.

Load Data

Create a sample 3-D surface referenced to geographic coordinates. Generate the *z*-coordinates by using the `peaks` function. Reference the *z*-coordinates to a region in latitude-longitude coordinates by using the `georefcells` function.

```
Z = 100*peaks;
RZ = georefcells([42.3342 42.3561],[-73.3191 -73.2847],size(Z));
```

To read spatially referenced raster data from a file, use the `readgeoraster` function.

Read Basemap Image

Specify latitude and longitude limits for the basemap image by expanding the 2-D limits of the surface. For this example, expand the limits by `0.01` degrees.

```
aoi = aoiquad(RZ);
aoi = buffer(aoi,0.01);
[latlim,lonlim] = bounds(aoi);
```

Read an image from the `"streets-light"` basemap as an array, a map cells reference object, and an attribution string.

```
[A,RA,attrib] = readBasemapImage("streets-light",latlim,lonlim);
```

Project Data to Web Mercator Coordinates

The `readBasemapImage` function spatially references basemap images by using a Web Mercator (WGS 84 / Pseudo-Mercator) projected CRS. To display the surface over the basemap image, you must project the geographic coordinates to the projected CRS.

Extract the geographic coordinates of the surface from the reference object. Then, project the geographic coordinates to Web Mercator coordinates.

```
[lat,lon] = geographicGrid(RZ);
[x,y] = projfwd(RA.ProjectedCRS,lat,lon);
```

Display Data over Basemap Image

Display the basemap image on a map with no axis labels.

```
figure
mapshow(A,RA)
hold on
axis off
```

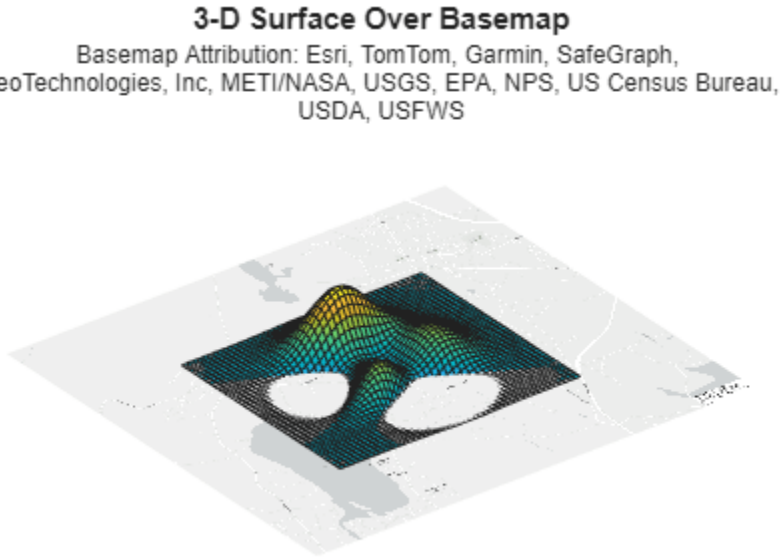
Display the 3-D surface on the same map.

```
surf(x,y,Z)
view(3)
```

Add a title and subtitle. Wrap the basemap attribution text.

```
title("3-D Surface Over Basemap")
```

```
t = "Basemap Attribution: " + attrib;
t = string(textwrap(t,70));
subtitle(t,FontSize=9)
```



See Also

Functions
`readBasemapImage` | `readgeoraster` | `projfwd`

Objects
`MapCellsReference` | GeographicAxes Properties

Related Examples

- “Types of MATLAB Plots”
- “Change Projection of Basemap Image” on page 4-59

Change Projection of Basemap Image

Geographic axes display data over basemaps using a Web Mercator projected coordinate reference system (CRS). While the Mercator projection preserves angles and is suitable for small regions, it distorts areas and is not appropriate for large or polar regions.

To display data over a basemap using a different projection method, you can read an image from a basemap by using the `readBasemapImage` function and then change the projection using one of these options:

- Change Projection Using Projection Stored in `axesm`-Based Map on page 4-59 — This option is convenient for projection methods supported by `axesm`-based maps.
- Change Projection Using Projected CRS Object on page 4-61 — This option is useful for projection methods not supported by `axesm`-based maps.

Change Projection Using Projection Stored in axesm-Based Map

This example shows how to project a basemap image by using the projection method stored in an `axesm`-based map. For more information about projections supported by `axesm`-based maps, see “Summary and Guide to Projections” on page 9-2.

Load Data

Read the latitude and longitude coordinates of European capitals from a text file.

```
[lat,lon] = readvars("european_capitals.txt");
```

Read Basemap Image

Specify latitude and longitude limits for the basemap image. Create a quadrangular area of interest (AOI) from the latitude and longitude coordinates. Buffer the AOI by 10 degrees. Then, find the latitude and longitude limits of the buffered AOI.

```
aoi = aoiquad(lat,lon);
aoi = buffer(aoi,10);
[latlim,lonlim] = bounds(aoi);
```

Read an image for the region from the `"bluegreen"` basemap, using a zoom level of `3`, as an array and a map cells reference object in Web Mercator coordinates.

```
[A,RA] = readBasemapImage("bluegreen",latlim,lonlim,3);
```

Unproject Basemap Image

Extract the Web Mercator coordinates of the basemap image from the reference object. Then, unproject the coordinates to geographic coordinates.

```
[xGrid,yGrid] = worldGrid(RA);
[latGrid,lonGrid] = projinv(RA.ProjectedCRS,xGrid,yGrid);
```

The coordinates of the European capitals are already in geographic coordinates. If you are adapting this example to your own code and your data is in projected coordinates, you must unproject your data to geographic coordinates.

Create Map

Create an `axesm`-based map with a projection that is suitable for the region by using the `worldmap` function. Alternatively, you can create an `axesm`-based map with a specified projection ID by using the `axesm` function.

```
figure
h = worldmap(latlim,lonlim);
```

View the projection method stored in the map.

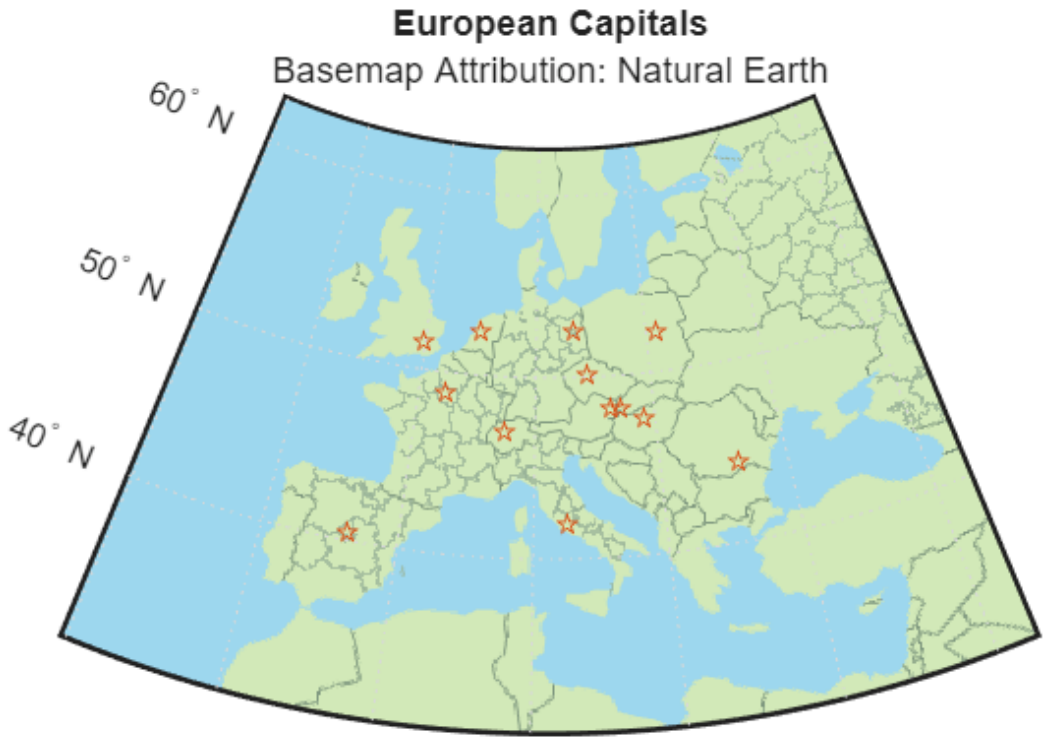
```
getm(h,"mapprojection")
```

```
ans =
'eqdconic'
```

The result `'eqdconic'` means that the map uses an Equidistant Conic projection.

Display the basemap and the European capitals on the map. The `geoshow` function projects and displays the latitude-longitude coordinates of the basemap image and capitals using the projection method stored in the map.

```
geoshow(latGrid,lonGrid,A)
geoshow(lat,lon,DisplayType="point",Marker="pentagram")
title("European Capitals")
subtitle("Basemap Attribution: Natural Earth")
```



Text in the projected basemap image can appear distorted.

Change Projection Using Projected CRS Object

This example shows how to read a basemap image using a projected CRS object and then display the projected image on an axes.

Load Data

Read a shapefile containing the coordinates of world cities into the workspace as a geospatial table. Extract the latitude and longitude coordinates.

```
GT = readgeotable("worldcities.shp");
lat = GT.Shape.Latitude;
lon = GT.Shape.Longitude;
```

Specify Projected CRS and Find Area of Use

Specify the projected CRS you want to use. For this example, create a `projcrs` object for WGS 84 / Equal Earth Asia-Pacific, which has the EPSG code 8859.

```
equalEarth = projcrs(8859);
```

Different projected CRSs are valid for different areas of use. For many projected CRSs, you can find the area of use by displaying the well-known text (WKT) string of the CRS and finding the BBOX attribute. This WKT specifies BBOX in the second-to-last line.

```
wktstring(equalEarth,Format="formatted")
```

```
ans =
    "PROJCRS["WGS 84 / Equal Earth Asia-Pacific",
         BASEGEOGCRS["WGS 84",
             ENSEMBLE["World Geodetic System 1984 ensemble",
                 MEMBER["World Geodetic System 1984 (Transit)"],
                 MEMBER["World Geodetic System 1984 (G730)"],
                 MEMBER["World Geodetic System 1984 (G873)"],
                 MEMBER["World Geodetic System 1984 (G1150)"],
                 MEMBER["World Geodetic System 1984 (G1674)"],
                 MEMBER["World Geodetic System 1984 (G1762)"],
                 MEMBER["World Geodetic System 1984 (G2139)"],
                 ELLIPSOID["WGS 84",6378137,298.257223563,
                     LENGTHUNIT["metre",1]],
                 ENSEMBLEACCURACY[2.0]],
             PRIMEM["Greenwich",0,
                 ANGLEUNIT["degree",0.0174532925199433]],
             ID["EPSG",4326]],
         CONVERSION["Equal Earth Asia-Pacific",
             METHOD["Equal Earth",
                 ID["EPSG",1078]],
             PARAMETER["Longitude of natural origin",150,
                 ANGLEUNIT["degree",0.0174532925199433],
                 ID["EPSG",8802]],
             PARAMETER["False easting",0,
                 LENGTHUNIT["metre",1],
                 ID["EPSG",8806]],
             PARAMETER["False northing",0,
                 LENGTHUNIT["metre",1],
                 ID["EPSG",8807]]],
         CS[Cartesian,2],
             AXIS["(E)",east,
```

```
        ORDER[1],
        LENGTHUNIT["metre",1]],
    AXIS["(N)",north,
        ORDER[2],
        LENGTHUNIT["metre",1]],
USAGE[
    SCOPE["Very small scale equal-area mapping - Asia-Pacific-centred."],
    AREA["World centred on Asia-Pacific."],
    BBOX[-90,-29.99,90,-30.01]],
ID["EPSG",8859]]"
```

Specify the latitude and longitude limits for the basemap image by using the information in `BBOX`. For this projected CRS, unwrap the maximum longitude by adding `360` degrees to the minimum longitude.

```
latlim = [-90 90];
lonmin = -29.99;
lonlim = [lonmin lonmin+360];
```

Read Basemap Image

Read an image from the `"satellite"` basemap as an array, a map cells reference object in Web Mercator coordinates, and an attribution string.

```
[A,RA,attrib] = readBasemapImage("satellite",latlim,lonlim);
```

Reproject Data and Basemap Image

Reproject the coordinates of the basemap image to Equal Earth coordinates.

- Extract the Web Mercator coordinates of the basemap image from the reference object.
- Unproject the Web Mercator coordinates to geographic coordinates.
- Project the geographic coordinates to Equal Earth coordinates.

```
[xGrid,yGrid] = worldGrid(RA);
[latGrid,lonGrid] = projinv(RA.ProjectedCRS,xGrid,yGrid);
[xEqualEarth,yEqualEarth] = projfwd(equalEarth,latGrid,lonGrid);
```

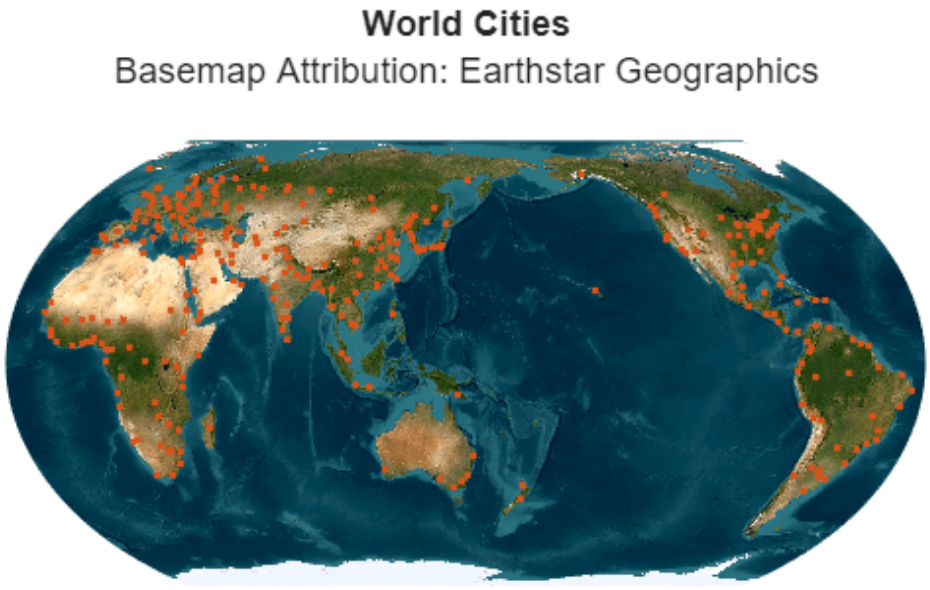
Project the geographic coordinates of the world cities to Equal Earth coordinates. If you are adapting this example to your own code and your data is in projected coordinates, you must first unproject your data to geographic coordinates.

```
[xData,yData] = projfwd(equalEarth,lat,lon);
```

Create Map

Display the projected basemap image and world cities on a map with no axis labels.

```
figure
mapshow(xEqualEarth,yEqualEarth,A)
hold on
mapshow(xData,yData,DisplayType="point",Marker=".")
title("World Cities")
subtitle("Basemap Attribution: " + attrib)
axis off
```



Text in the projected basemap image can appear distorted.

See Also

Functions
`readBasemapImage` | `worldmap` | `geoshow` | `mapshow` | `projfwd` | `projinv`

Objects
`MapCellsReference` | `projcrs`

Related Examples

- “Create Common Plots over Basemap Images” on page 4-52

Use Basemaps in Offline Environments

Provide geographic context for your maps by displaying data over a basemap, also called a base layer. MathWorks provides a variety of basemaps, including low-resolution two-tone options created using Natural Earth and high-zoom-level options hosted by Esri®. For a list of basemap options, see the `basemap` argument of the `geobasemap` function.

Most of these basemaps require internet access, including the default basemap. If you do not have reliable access to the internet, you can use strategies such as downloading a selection of basemaps onto your local system or displaying data over the `"darkwater"` basemap. Some strategies require MATLAB to have temporary internet access and other strategies do not require MATLAB to have internet access. Different strategies support different types of map displays.

Internet Access Requirement for MATLAB	Strategy	Supported Map Displays

"Temporary Internet Access" on page 4-65	Download basemaps onto your local system on page 4-0	• Objects with a `Basemap` property, for example:
		  • Geographic axes
		  • Geographic globes
		  • Site Viewer (Communications Toolbox™, Antenna Toolbox™)
		  • Satellite Scenario Viewer (Satellite Communications Toolbox)

	Use cached basemap tiles on page 4-0	• Geographic axes • Geographic bubble charts
	Use a spatially referenced image from a web source on page 4-0	• Axes (`mapshow`) • `axesm`-based maps (`geoshow`)
"No Internet Access" on page 4-68	Create a basemap from an MBTiles file on page 4-0	• Objects with a `Basemap` property
	Create a basemap from an image on page 4-0	• Objects with a `Basemap` property
	Use a basemap installed with MATLAB® on page 4-0	• Objects with a `Basemap` property
	Use a spatially referenced image from a file on page 4-0	• Axes (`mapshow`) • `axesm`-based maps (`geoshow`)
	Use spatially referenced vector data from a file on page 4-0	• Map axes (`geoplot`) • Geographic axes (`geoplot`) • Axes (`mapshow`) • `axesm`-based maps (`geoshow`)

Temporary Internet Access

These strategies require MATLAB to temporarily access the internet. When MATLAB is in an online environment, you can save the basemap data to your local system. Then, when MATLAB is in an offline environment, you can load the saved basemap data.

- Download basemaps onto your local system on page 4-65
- Use cached basemap tiles on page 4-65
- Use a spatially referenced image from a web source on page 4-66

Download basemaps onto your local system

When you have internet access, you can download the basemaps created using Natural Earth onto your local system. The basemaps hosted by Esri are not available for download. Then, when you are in an offline environment, you can use the downloaded basemaps.

Download basemaps using the Add-On Explorer.

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** and then **Get Add-Ons**.
2. In the **Add-On Explorer**, scroll to the **MathWorks Optional Features** section, and click **Show All** to find the basemap add-ons. You can also search for the basemap add-ons by name, or click **Optional Features** in **Filter by Type**.
3. Select the basemap add-ons that you want to download.

These are the names of the basemap add-ons and their associated basemap argument names.

- MATLAB Basemap Data - bluegreen (`"bluegreen"`)
- MATLAB Basemap Data - grayland (`"grayland"`)
- MATLAB Basemap Data - colorterrain (`"colorterrain"`)
- MATLAB Basemap Data - grayterrain (`"grayterrain"`)
- MATLAB Basemap Data - landcover (`"landcover"`)

Use cached basemap tiles

When you view a basemap over the internet, MATLAB temporarily caches the basemap tiles. If you lose your internet connection, you can still view the cached basemap tiles.

When you are in an offline environment and attempt to view basemap tiles that you have not previously viewed, MATLAB issues a warning and displays different tiles depending on the type of basemap. When you attempt to view tiles created using Natural Earth, MATLAB displays `"darkwater"` tiles. When you attempt to view tiles from a custom basemap or tiles hosted by Esri, you see missing map tiles.

Cached basemap tiles have these limitations:

- MATLAB stores the cached tiles in a temporary folder. Your operating system can delete the temporary folder.
- The cached tiles can expire.
- MATLAB caches custom basemap tiles for only the current MATLAB session.

Use a spatially referenced image from a web source

When you have internet access, you can read an image from a basemap or a Web Map Service (WMS) server as a spatially referenced image and save the image as a MAT or GeoTIFF file. Then, when you are in an offline environment, you can load the image from the file and use it as a base layer for your data.

Read and save an image from a basemap

Read satellite imagery for Boston into the workspace as an array, a map cells reference object, and an attribution by using the `readBasemapImage` function.

```
[A1,R1,attrib1] = readBasemapImage("satellite",[42.3453 42.3711],[-71.099 -71.0454]);
```

Save the imagery as a GeoTIFF file by using the `geotiffwrite` function.

- The `readBasemapImage` function spatially references the image using the Web Mercator projected coordinate reference system (CRS) with EPSG code `3857`. Specify the CRS code by using the `CoordRefSysCode` name-value argument.
- Include the attribution by specifying the `ImageDescription` TIFF tag.

```
tag = struct("ImageDescription",attrib1);
geotiffwrite("satelliteBoston.tif",A1,R1,CoordRefSysCode=3857,TiffTags=tag)
```

To use the basemap data, load the GeoTIFF file and attribution.

```
[A2,R2] = readgeoraster("satelliteBoston.tif");
info = imfinfo("satelliteBoston.tif");
attrib2 = info.ImageDescription;
```

Display the data on a map. Use the `mapshow` function when the reference object is a map reference object and use the `geoshow` function when the reference object is a geographic reference object.

```
figure
mapshow(A2,R2)
title("Boston Imagery from Basemap")
subtitle("Attribution: " + attrib2)
axis off
```

For more information about displaying data over basemaps images in offline environments, see “Display Data over Basemaps When Internet Access Is Intermittent” on page 4-79.

Read and save an image from a WMS server

WMS servers provide publicly accessible geospatial data from web-based sources. Search the WMS Database for orthoimagery from the USGS National Map.

```
usgs = wmsfind("basemap.nationalmap",SearchFields="ServerURL");
imagery = refine(usgs,"USGSImageryOnly",SearchFields="ServerURL");
imagery = wmsupdate(imagery);
```

Read orthoimagery for an area in Boston and save it as a GeoTIFF file.

```
[A3,R3] = wmsread(imagery,Latlim=[42.3453 42.3711],Lonlim=[-71.099 -71.0454]);
geotiffwrite("usgsImageryLayer.tif",A3,R3)
```

To use the WMS orthoimagery, load the GeoTIFF file and display it on a map.

```
[A4,R4] = readgeoraster("usgsImageryLayer.tif");
figure
usamap(A4,R4)
geoshow(A4,R4)
title("Boston Imagery from WMS Server")
subtitle("Attribution: USGS National Map")
```

For more information about creating WMS maps in offline environments, see “Create WMS Maps When Internet Access Is Intermittent” on page 7-6.

No Internet Access

These strategies do not require MATLAB to access the internet.

- Create a basemap from an MBTiles file on page 4-68
- Create a basemap from an image on page 4-70
- Use `"darkwater"` basemap installed with MATLAB on page 4-71
- Use a spatially referenced image from a file on page 4-72
- Use spatially referenced vector data from a file on page 4-73

Create a basemap from an MBTiles file

When you are in an offline environment, you can use the `addCustomBasemap` function to create a custom basemap from an MBTiles file.

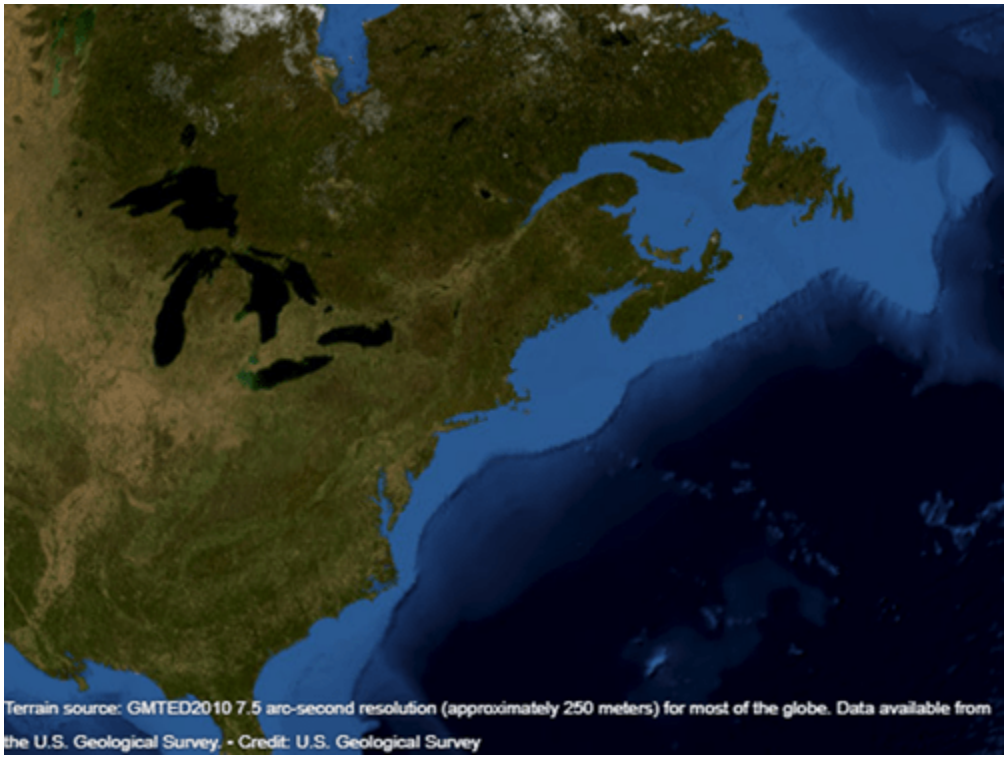
MBTiles files can contain raster or vector map tiles. If you do not know whether your MBTiles file contains raster or vector map tiles, check with your data provider.

- Raster map tiles are useful for displaying satellite imagery and orthoimagery. You can write raster data to an MBTiles file by using the `mbtileswrite` function.
- Vector map tiles enable you to customize the appearance of the basemap and typically include higher zoom levels than raster map tiles. For information about customizing the appearance of vector basemaps, see “Customize Appearance of Vector Basemaps” on page 4-45.

Create a basemap from a raster MBTiles file

Mapping Toolbox™ includes a raster MBTiles file with low-resolution USGS imagery called `usgsimagery.mbtiles`. Create a custom basemap from the file and display it using a geographic globe. Position the camera 3200 kilometers above Boston.

```
addCustomBasemap("usgsimagery","usgsimagery.mbtiles")

uif = uifigure;
g = geoglobe(uif);
geobasemap(g,"usgsimagery")
campos(g,42.3582,-71.0722,3200000)
```



Create a basemap from a vector MBTiles file

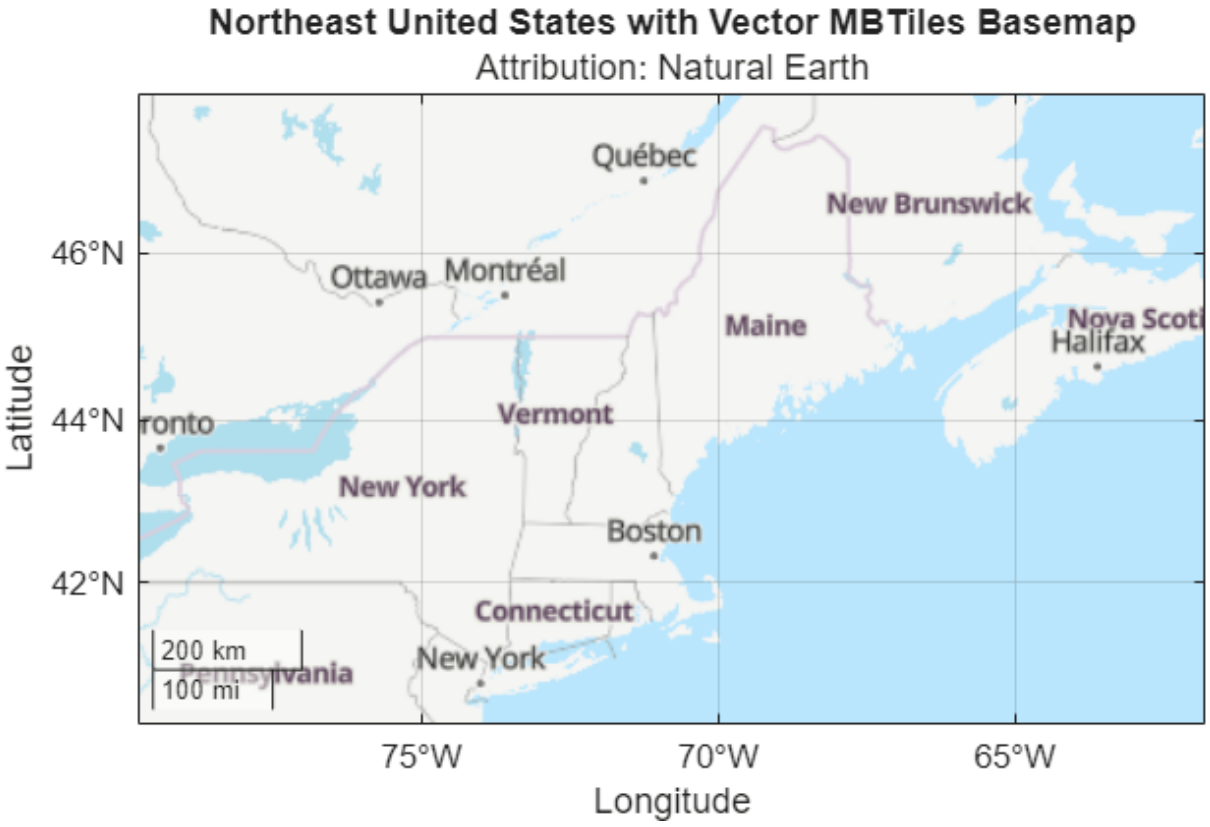
(Since R2023a)

Mapping Toolbox includes a vector MBTiles file, `naturalearth.mbtiles`, with low-zoom levels of region and land boundaries. Create a custom basemap from the file. Specify the appearance of the basemap, including the colors and fonts, using a predefined topographic style. Predefined styles do not require internet access. For information about vector basemap styles, including limitations in offline environments, see the “Style” argument on the `addCustomBasemap` reference page.

```
addCustomBasemap("naturalearth","naturalearth.mbtiles",Style="topographic")
```

Display the custom basemap using a geographic axes.

```
figure
geobasemap naturalearth
geolimits([40.1951 47.8433],[-77.5238 -64.0377])
title("Northeast United States with Vector MBTiles Basemap")
subtitle("Attribution: Natural Earth")
```



Create a basemap from an image

When you are in an offline environment, you can use the `addCustomBasemap` function to create a custom basemap from a spatially referenced raster image. The function you use to read the image into the workspace depends on the file format.

- Read images in formats such as GeoTIFF by using the `readgeoraster` function.
- Read images in formats that are associated with a world file, such as TIFF, JPEG, or PNG, by using the `imread` and `worldfileread` functions.

Read a GeoTIFF image of Boston into the workspace as an array and a map cells reference object. The array is a grid of cells referenced to projected coordinates.

```
[A5,R5] = readgeoraster("boston.tif");
```

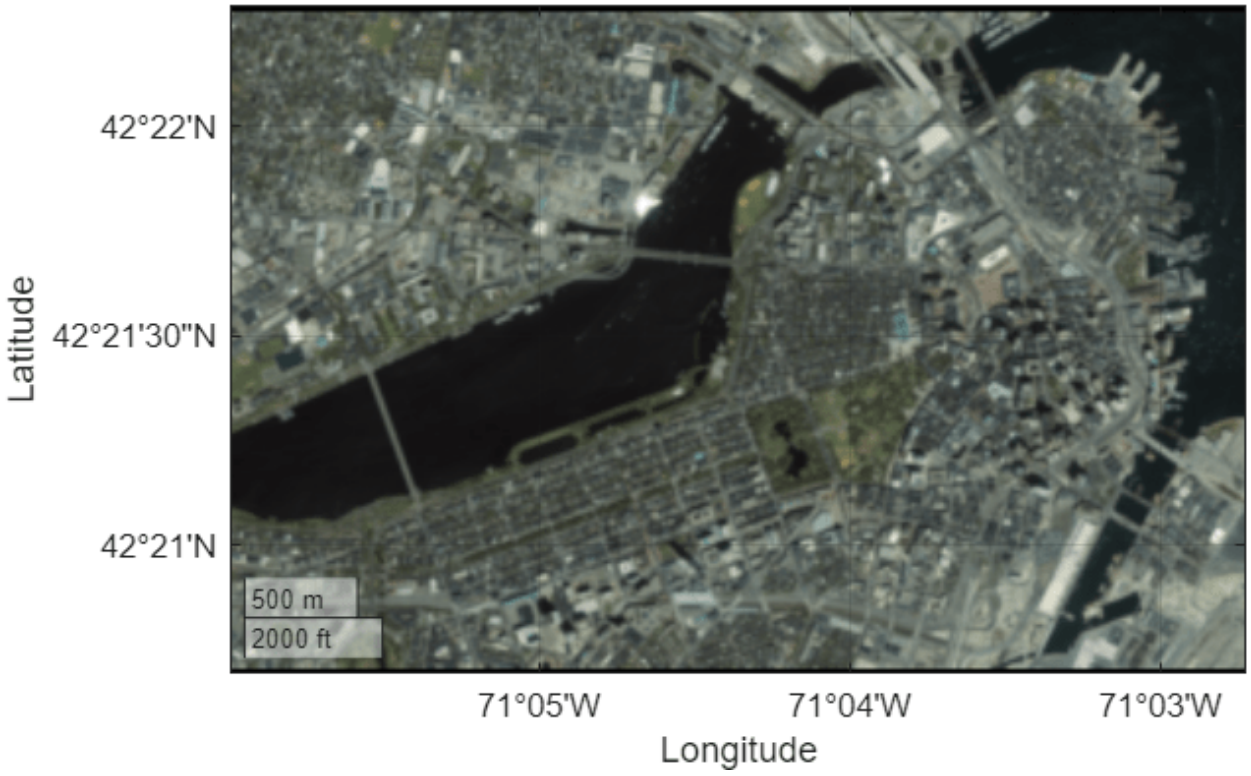
Add a custom basemap from the array and reference object. The `addCustomBasemap` function issues a warning because the array and reference object do not specify global data for zoom level 0.

```
addCustomBasemap("bostonBasemap",A5,R5)
```

```
Warning: Basemap contains regional data with latitude limits in the range [42.3451, 42.3713] and
```

Display the custom basemap using a geographic axes. MATLAB sets the limits of the axes to closely match the limits of the basemap data. The black background indicates areas where the basemap tile contains no data.

```
figure
geobasemap bostonBasemap
```

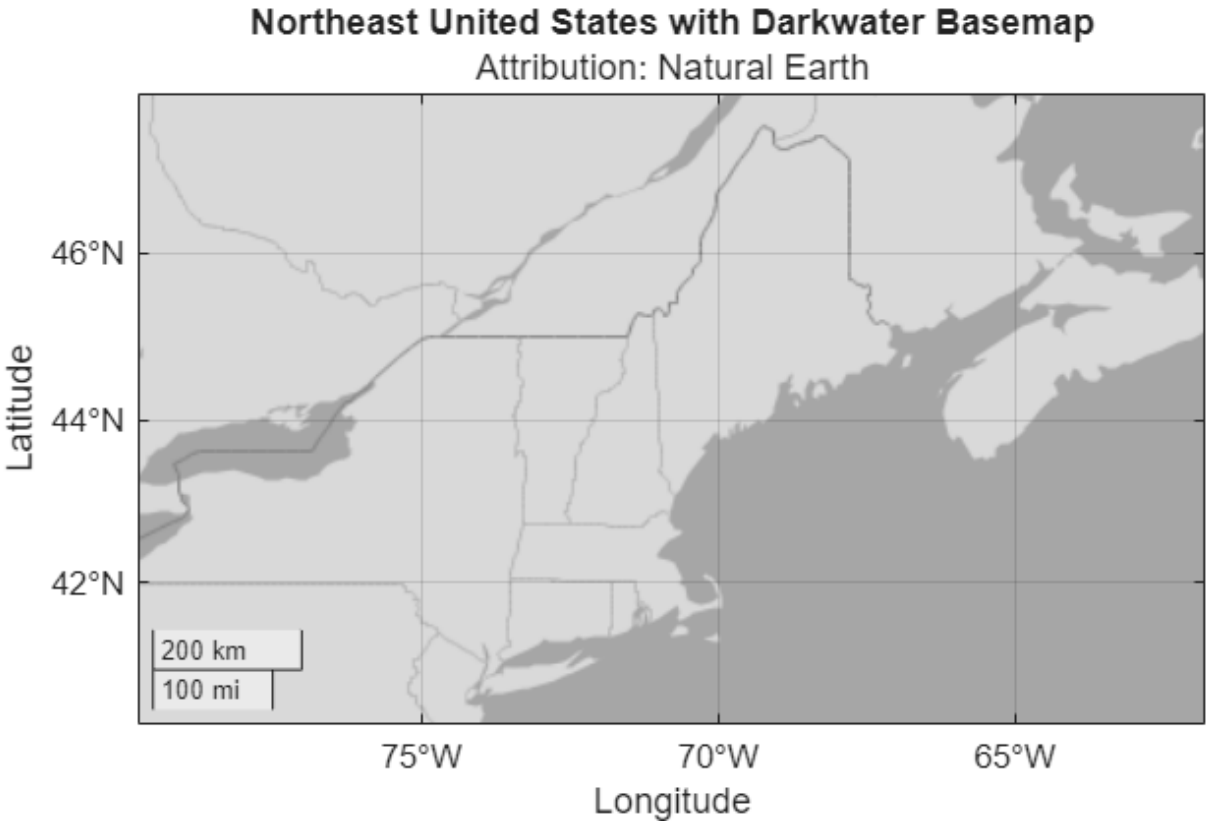


Use "darkwater" basemap installed with MATLAB

MATLAB includes the "darkwater" basemap, a two-tone land-ocean map with light gray land areas and dark grey water areas. Using the "darkwater" basemap does not require internet access.

Display the "darkwater" basemap using a geographic axes. Center the map over the northeast United States.

```
figure
geobasemap darkwater
geolimits([40.1951 47.8433],[-77.5238 -64.0377])
title("Northeast United States with Darkwater Basemap")
subtitle("Attribution: Natural Earth")
```



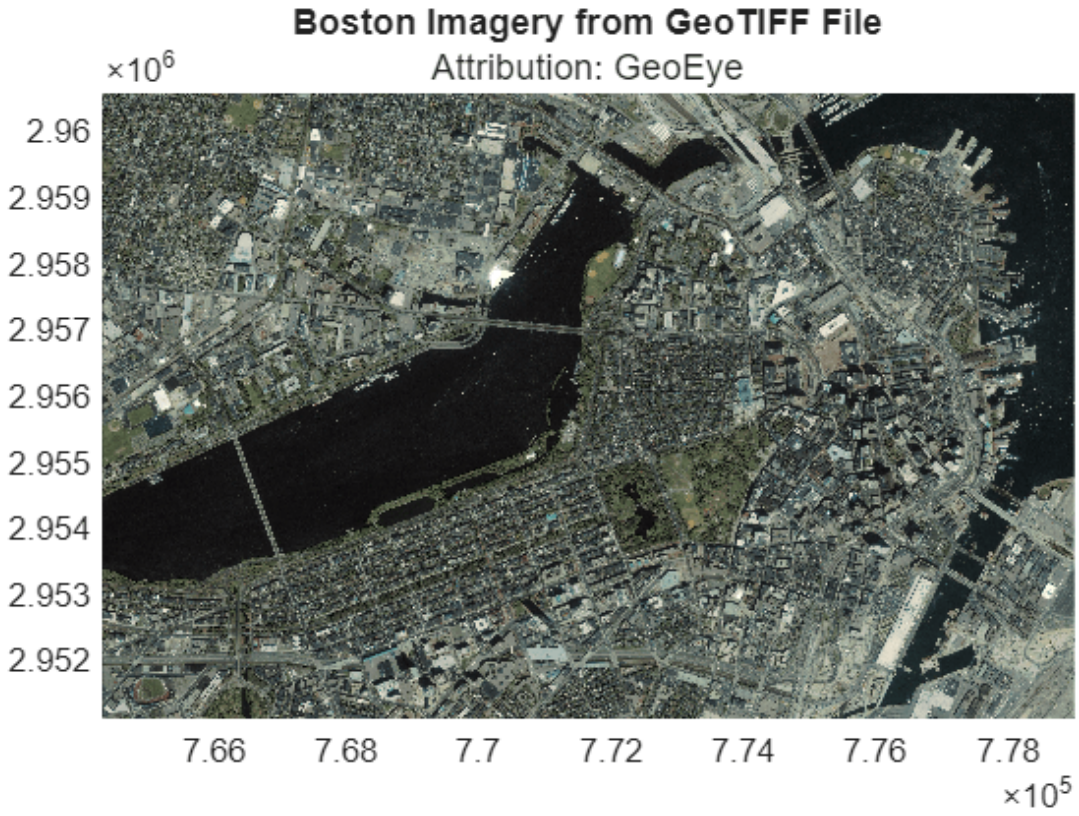
Use a spatially referenced image from a file

When you are in an offline environment, you can read a spatially referenced raster image from a file and then use it as a base layer for your data. The function you use to read the data depends on the file format.

- Read images in formats such as GeoTIFF by using the `readgeoraster` function.
- Read images in formats that are associated with a world file, such as TIFF, JPEG, or PNG, by using the `imread` and `worldfileread` functions.

Read a GeoTIFF image of Boston into the workspace as an array and a map cells reference object. Then, display the image on a map. Use the `mapshow` function when the reference object is a map reference object and use the `geoshow` function when the reference object is a geographic reference object.

```
[A6,R6] = readgeoraster("boston.tif");
figure
mapshow(A6,R6)
title("Boston Imagery from GeoTIFF File")
subtitle("Attribution: GeoEye")
```

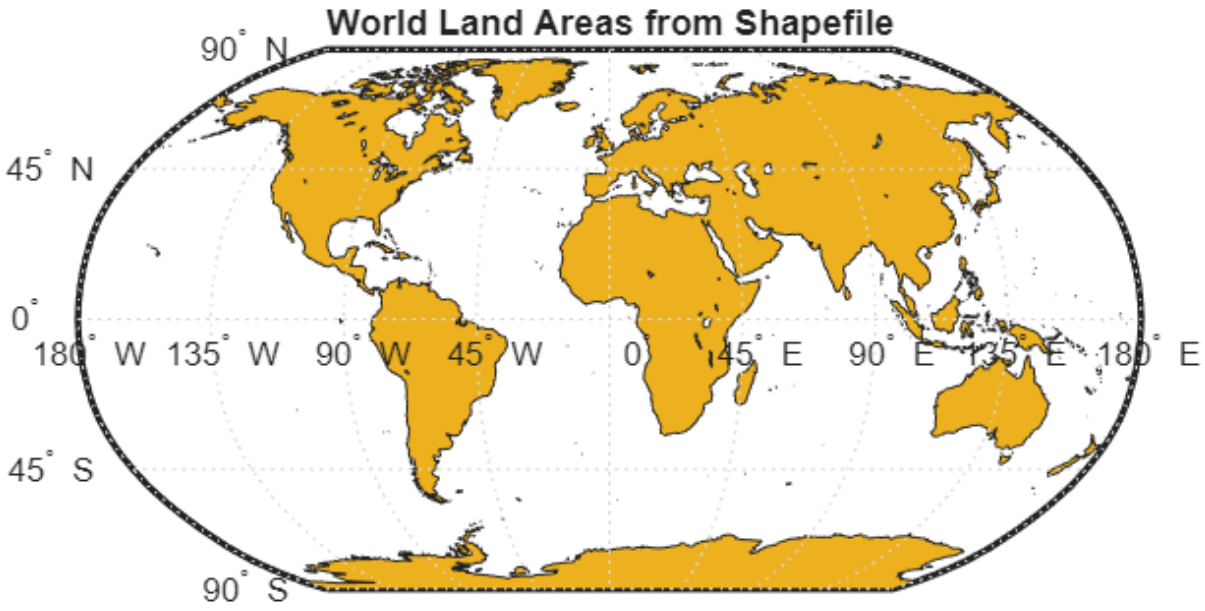


Use spatially referenced vector data from a file

When you are in an offline environment, you can read spatially referenced vector data from a file and then use it as a base layer for your data. Read vector data in formats such as shapefile by using the `readgeotable` function.

Mapping Toolbox™ includes a shapefile with world land areas called `landareas.shp`. Read the land areas into the workspace as a geospatial table. Then, display the land areas on a world map.

```
GT = readgeotable("landareas.shp");

figure
worldmap world
geoshow(GT)
title("World Land Areas from Shapefile")
```

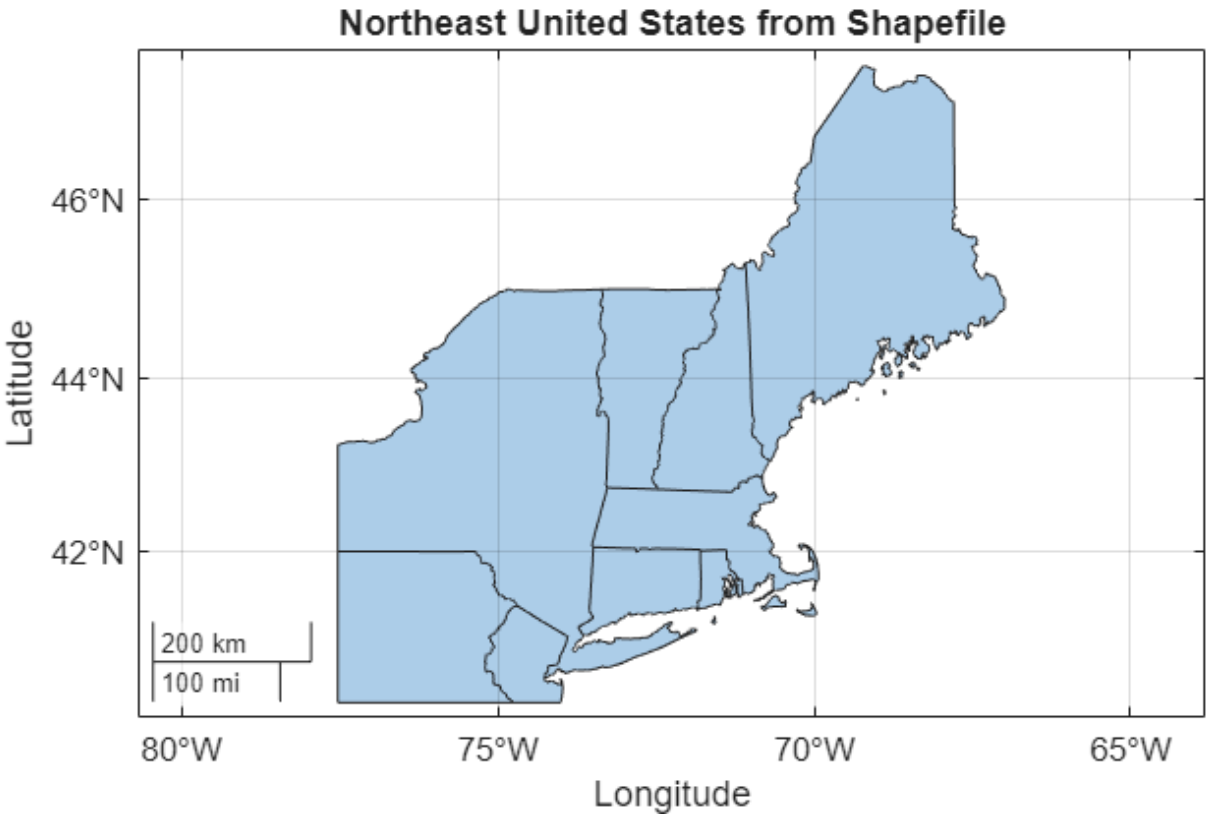


Mapping Toolbox™ includes shapefiles with high- and low-resolution US state boundaries called `usastatehi.shp` and `usastatelo.shp`, respectively. Read the high-resolution state boundaries into the workspace as a geospatial table. Clip the boundaries to include only the northeast states.

```
GT = readgeotable("usastatehi.shp");
ne = geoclip(GT.Shape,[40.1951 47.8433],[-77.5238 -64.0377]);
```

Display the boundaries using a geographic axes. To prevent MATLAB from using the basemaps provided by MathWorks, set the basemap to `"none"`.

```
figure
geobasemap none
hold on
geoplot(ne)
title("Northeast United States from Shapefile")
```



For an example that shows how to create a base layer from data stored in an OpenStreetMap® file, see “Create Base Layer from OpenStreetMap Data” on page 4-76.

See Also

Functions

`geobasemap` | `geotiffwrite` | `wmsread`

Related Examples

- “Access Basemaps for Geographic Axes and Charts”
- “Access Basemaps and Terrain for Geographic Globe” on page 5-2

Create Base Layer from OpenStreetMap Data

When you are in an offline environment, you can create provide geographic context for your data by creating a simple base layer from data stored in an OpenStreetMap® file. This example shows how to create a base layer for several city blocks in Shibuya, Tokyo, Japan by reading building, road, and traffic signal data from the file.

Read Building Data

Read the buildings layer from an OpenStreetMap file [1] on page 4-78 by using the `readgeotable` function. The function derives building information from the file and stores the result in a geospatial table. The table represents the buildings using polygon shapes in geographic coordinates.

```
filename = "shibuya.osm";
buildingsLayer = readgeotable(filename,Layer="buildings");
```

Read Road Data

The lines layer of an OpenStreetMap file represents line-like features such as roads, sidewalks, and subway routes. Read the lines layer from the file into a geospatial table. The table represents the lines using polygon shapes in geographic coordinates.

```
linesLayer = readgeotable(filename,Layer="lines");
```

A table row represents a principal road when the `highway` table variable has a value of `"motorway"`, `"trunk"`, `"primary"`, `"secondary"`, `"tertiary"`, or `"residential"`. Create a new geospatial table from the table rows that represent principal roads.

```
roadtypes = ["motorway","trunk","primary","secondary","tertiary","residential"];
roadRows = ismember(linesLayer.highway,roadtypes);
roads = linesLayer(roadRows,:);
```

Read Traffic Signal Data

The points layer of an OpenStreetMap file represents point-like features such as traffic signals, bus stops, and subway entrances. Read the points layer from the file into a geospatial table. The table represents the points using point shapes in geographic coordinates.

```
pointsLayer = readgeotable(filename,Layer="points");
```

A table row represents a traffic signal when the `highway` table variable has a value of `"traffic_signals"`. Create a new geospatial table from the table rows that represent traffic signals.

```
signalRows = ismember(pointsLayer.highway,"traffic_signals");
signals = pointsLayer(signalRows,:);
```

Prepare to display the traffic signals using an icon chart by extracting the latitude and longitude coordinates from the table.

```
signalLat = signals.Shape.Latitude;
signalLon = signals.Shape.Longitude;
```

Display Base Layer

Set up a new map using a projected coordinate reference system (CRS) that is appropriate for Japan. Provide geographic context for the area by displaying the buildings, roads, and traffic signals.

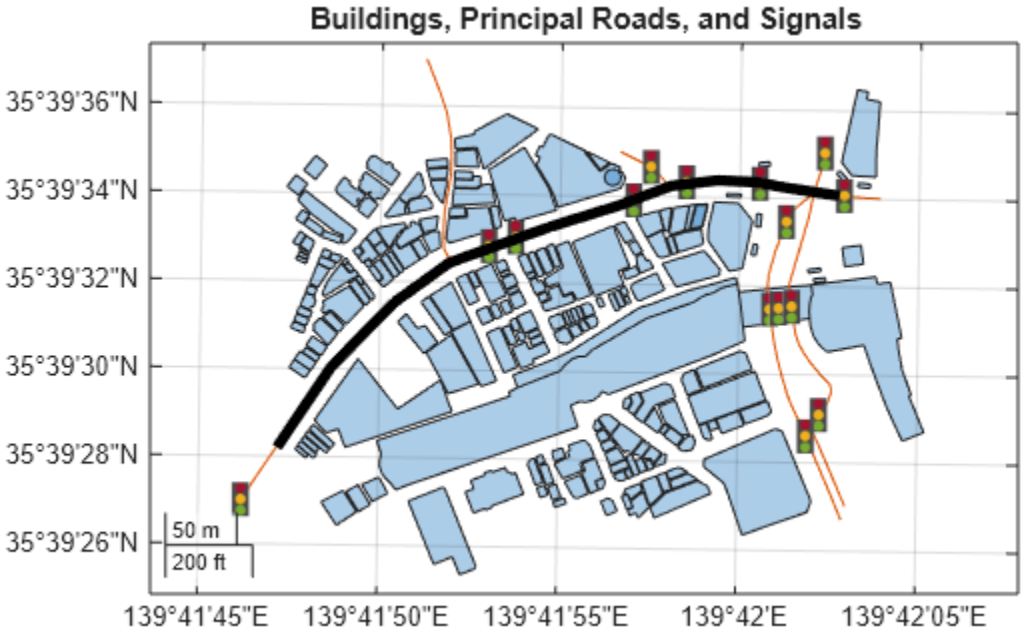
Prevent the objects from capturing mouse clicks by setting the `PickableParts` properties to `"none"`.

```
figure
proj = projcrs(32654);
newmap(proj)

geoplot(buildingsLayer,PickableParts="none")
hold on
geoplot(roads,PickableParts="none")
geoiconchart(signalLat,signalLon,"traffic-signal.png",SizeData=17,PickableParts="none")
title("Buildings, Principal Roads, and Signals")
```

Load and Plot Data

Read a driving track from a GPX file into a geospatial table. Extract the coordinates from the table, Then, display the track on the map using a thick, black line.

```
trk = readgeotable("shibuya_track.gpx",Layer="track_points");
geoplot(trk.Shape.Latitude,trk.Shape.Longitude,"k",LineWidth=4)
```



[1] You can download OpenStreetMap files from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

See Also

Functions
`readgeotable` | `geoplot`

Related Examples

- “Read Data from OpenStreetMap Files” on page 2-21
- “Display Buildings from OpenStreetMap Files” on page 2-27
- “Use Basemaps in Offline Environments” on page 4-64

Display Data over Basemaps When Internet Access Is Intermittent

In most cases, displaying data over basemaps requires internet access. When you have intermittent internet access, you can prepare for an offline workflow by reading an image from a basemap and saving the image as a MAT or GeoTIFF file. Then, when you are in an offline environment, you can load the image from the file and use it as a base layer for your data.

This example shows how to plot point data over a basemap image. The behavior of the example depends on whether you have internet access, but the resulting map is the same.

Specify whether you have internet access by using the logical scalar `useInternet`.

- If `useInternet` is `true`, read an image from the basemap and save it to a GeoTIFF file.
- If `useInternet` is `false`, read an image from a GeoTIFF file.

```
useInternet = true;
```

If you use this example as a model to create your own maps, you must create the saved files the first time you run the example by setting `useInternet` to `true`.

Load Data

Import a shapefile containing the coordinates of locations in Boston as a geospatial table. The shapefile represents the locations using points in a projected coordinate reference system (CRS). Extract the *x*,*y*-coordinates.

```
GT = readgeotable("boston_placenames.shp");
pointsX = GT.Shape.X;
pointsY = GT.Shape.Y;
```

Get information about the projected CRS.

```
pointsCRS = GT.Shape.ProjectedCRS
```

```
pointsCRS =
  projcrs with properties:

                    Name: "NAD83 / Massachusetts Mainland"
           GeographicCRS: [1×1 geocrs]
        ProjectionMethod: "Lambert Conic Conformal (2SP)"
              LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Read Basemap

When you read an image from a basemap using the `readBasemapImage` function, you must specify the region to read using latitude and longitude coordinates. For this example, however, the locations are in projected coordinates. Unproject the *x*,*y*-coordinates to latitude and longitude coordinates.

```
[pointslat,pointslon] = projinv(pointsCRS,pointsX,pointsY);
```

Create a quadrangular area of interest (AOI) from the latitude and longitude coordinates. Buffer the AOI by 0.005 degrees. Then, find the latitude and longitude limits of the buffered AOI.

```
aoi = aoiquad(pointslat,pointslon);
aoi = buffer(aoi,0.005);
[latlim,lonlim] = bounds(aoi);
```

Read a basemap image for the region from the `"satellite"` basemap.

- If `useInternet` is `true`, read the basemap image as an array, a map cells reference object in Web Mercator coordinates (EPSG code 3857), and an attribution string. Save the array and reference object to a GeoTIFF file. Include the attribution by specifying the `ImageDescription` TIFF tag.
- If `useInternet` is `false`, load the basemap image and attribution.

```
if useInternet
    [A,R,attrib] = readBasemapImage("satellite",latlim,lonlim);
    tag = struct("ImageDescription",attrib);
    geotiffwrite("satelliteBoston.tif",A,R,CoordRefSysCode=3857,TiffTags=tag)
else
    [A,R] = readgeoraster("satelliteBoston.tif");
    info = imfinfo("satelliteBoston.tif");
    attrib = info.ImageDescription;
end
```

Project Data to Web Mercator Coordinates

Project the geographic coordinates to the same CRS as the basemap image.

```
[wmX,wmY] = projfwd(R.ProjectedCRS,pointslat,pointslon);
```

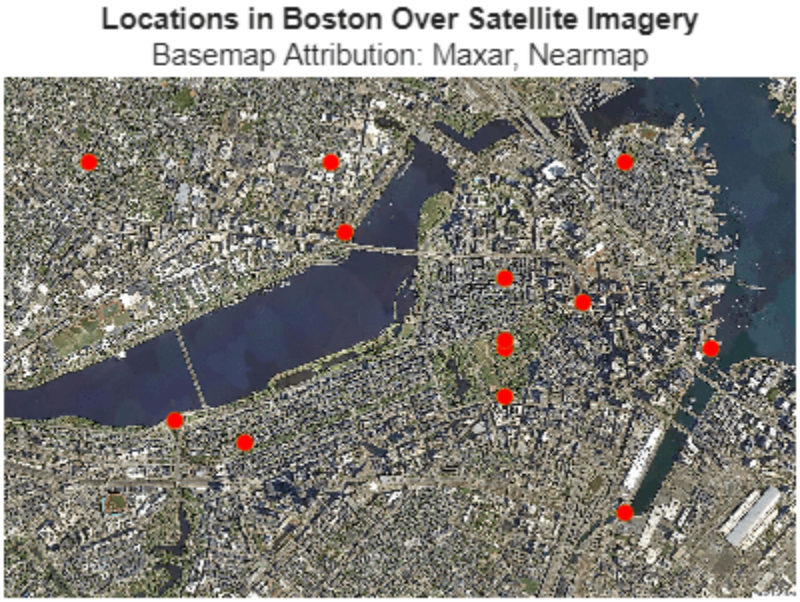
Display Data over Basemap Image

Display the basemap image on a map with no axis labels.

```
figure
mapshow(A,R)
hold on
axis off
```

Display the locations on the same map.

```
mapshow(wmX,wmY,DisplayType="point",Marker="o",MarkerFaceColor="r")
title("Locations in Boston Over Satellite Imagery")
subtitle("Basemap Attribution: " + attrib)
```



The geographic CRS underlying the geographic coordinates is NAD83, while the geographic CRS underlying the basemap image is WGS84. NAD83 and WGS84 are similar, but not identical. As a result, the coordinates and basemap can appear misaligned.

See Also

Functions
`readBasemapImage` | `geotiffwrite` | `readgeoraster`

Objects
`MapCellsReference` | `projcrs`

Related Examples

- “Use Basemaps in Offline Environments” on page 4-64
- “Create WMS Maps When Internet Access Is Intermittent” on page 7-6

Thematic Maps

Most published and online maps fall into four categories:

- Navigation maps, including topographic maps and nautical and aeronautical charts
- Geophysical maps, that show the structure and dynamics of earth, oceans and atmosphere
- Location maps, that depict the locations and names of physical features
- Thematic maps, that portray attribute data about locations and features

Although online maps often combine these categories in new and unexpected ways, published maps and atlases tend to respect them.

Thematic maps tend to be more highly stylized than other types of maps and frequently omit locational information such as place names, physical features, coordinate grids, and map scales. This is because rather than showing physical features on the ground, such as shorelines, roads, settlements, topography, and vegetation, a thematic map displays quantified facts (a "theme"), such as statistics for a region or sets of regions. Examples include the locations of traffic accidents in a city, or election results by state. Thematic maps have a wide vocabulary of cartographic symbols, such as point symbols, dot distributions, "quiver" vectors, isolines, colored zones, raised prisms, and continuous 3-D surfaces.

Choropleth and Classification Maps

Choropleth and classification maps use colors or patterns to represent attributes associated with certain geographic regions.

- Choropleth maps indicate the values of numeric attributes within spatial regions by using colors along a gradient. Choropleth maps are commonly used to illustrate population density within specified regions. For examples of how to create choropleth maps, see “Create Choropleth Map of Population Density” on page 4-85 and “Create Choropleth Map from Table Data” on page 4-88.
- Classification maps indicate the values of classified or categorical attributes within spatial regions by using discrete colors. Classification maps are commonly used to illustrate climate classes, land cover classes, and binned population density within specified regions. For an example of how to create a classification map, see “Create Classification Map from Table Data” on page 4-92.

Scatter and Bubble Maps

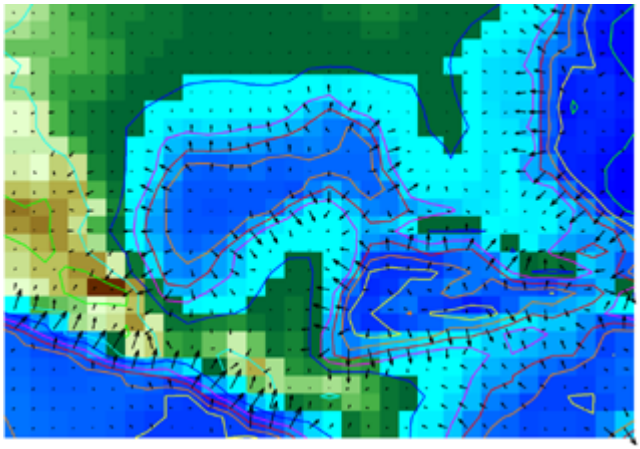
Scatter and bubble plots display symbols at specified point locations. If the symbols are small and inconspicuous and do not vary in size, the result is a *dot-distribution map*. If the symbols vary in size and/or shape according to a vector of attribute values, the result is a *proportional symbol map*.

You can create scatter and bubble plots in these ways:

- Create scatter plots in map axes and geographic axes by using the `geoscatter` function.
- Create scatter plots in `axesm`-based maps by using the `scatterm` function.
- Create bubble plots in map axes and geographic axes by using the `bubblechart` function.
- Create a geographic bubble chart by using the `geobubble` function.

Contour Maps

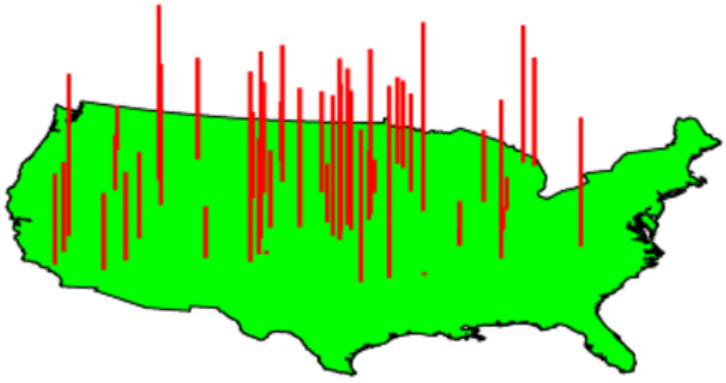
Contour and quiver plots can be useful in analyzing matrix data. This image shows contour elevation lines and quiver plots over a topographical map of the Gulf of Mexico. The quiver plots illustrate the gradient of the topographical matrix.



Create contour plots by using the `contourm`, `contour3m`, or `contourfm` function. Create 2-D or 3-D quiver plots over `axesm`-based maps by using the `quiverm` or `quiver3m` function.

Stem Maps

Stem plots are 3-D geographic bar graphs portraying numeric attributes at point locations, usually on vector base maps. Below is an example of a stem plot over a map of the continental United States. The bars could represent anything from selected city populations to the number of units of a product purchased at each location.



You can create stem plots over `axesm`-based maps by using the `stem3m` function.

See Also

Functions
`geoplot`

Related Examples

- “Create Common Plots over Basemap Images” on page 4-52

Create Choropleth Map of Population Density

Create a choropleth map of population density for several US states in the year 2000. A *choropleth map* indicates the values of numeric attributes within spatial regions by using colors along a gradient.

This page shows how to create similar maps using map axes (*since R2023a*) and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Prepare Data

Prepare the data to use in the examples.

Import a shapefile containing population density data for each US state. The shapefile represents the states using polygon shapes in geographic coordinates.

```
states = readgeotable("usastatelo.shp");
```

Create a subtable that includes New York and the states that comprise New England: Maine, New Hampshire, Vermont, Massachusetts, Connecticut, and Rhode Island.

```
names = ["New York" "Maine" "New Hampshire" "Vermont" "Massachusetts" ...
    "Connecticut" "Rhode Island"];
NEstates = geocode(names,states);
```

Create Map Using Map Axes

Create a choropleth map using a map axes object.

Set up a map using a projected coordinate reference system (CRS) that is appropriate for the conterminous United States. Create the CRS using the EPSG code `26918`, which uses a Transverse Mercator projection.

```
figure
proj = projcrs(26918);
newmap(proj)
```

Display the state polygons on the map. Specify the colors using the population data in the table.

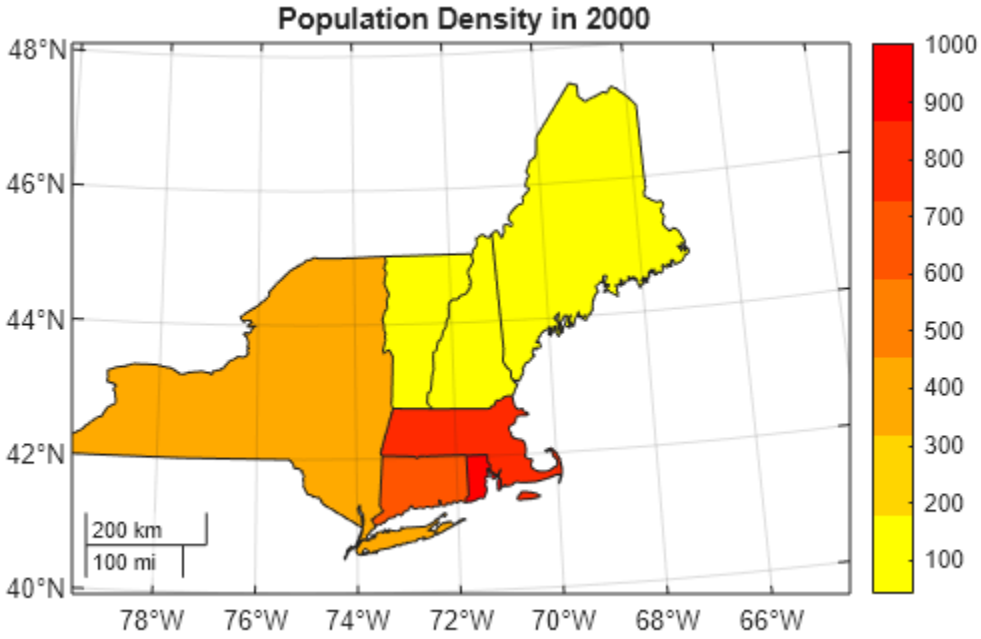
```
geoplot(NEstates,ColorVariable="PopDens2000")
```

Create a colormap with colors that transition from yellow to red. Apply the colormap to the map axes. Then, add a color bar and a title.

```
cmap = flipud(autumn(height(NEstates)));
colormap(cmap)

colorbar
title("Population Density in 2000")
```

Adjust the geographic limits.

```
geolimits([40 48],[-76 -68])
```



Create Map Using axesm-Based Map

Create a map using a projection that is appropriate for the conterminous United States. Use an Albers Equal Area Conic projection, and specify latitude and longitude limits for a region in the northeast United States.

```
figure
axesm(MapProjection="eqaconic",MapParallels=[], ...
  MapLatLimit=[41 48],MapLonLimit=[-74 -66])
```

Assign colors to the polygons based on the population density by creating a symbol specification. Specify the colors using a colormap that transitions from yellow to red.

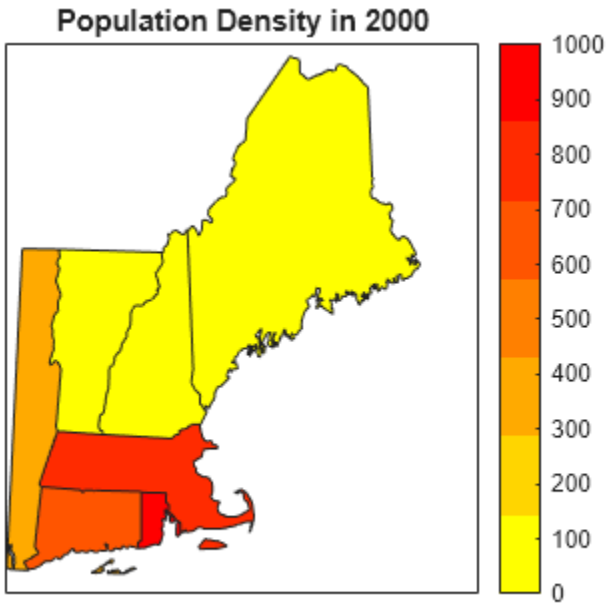
```
maxdensity = max([NEstates.PopDens2000]);
cmap = flipud(autumn(height(NEstates)));
polyColors = makesymbolspec("Polygon", ...
    {"PopDens2000",[0 maxdensity],"FaceColor",cmap});
```

Display the state polygons on the map. Specify the colors using the symbol specification.

```
geoshow(NEstates,SymbolSpec=polyColors)
```

Apply the colormap to the axes. Then, add a color bar and a title.

```
colormap(cmap)

colorbar
clim([0 maxdensity])
title("Population Density in 2000")
```



See Also

Functions
`newmap` | `geoplot` | `axesm` | `geoshow` | `colormap`

Related Examples

- “Create Choropleth Map from Table Data” on page 4-88
- “Create Classification Map from Table Data” on page 4-92
- “Create Map of Quadrangle Using Cartographic Map Layout” on page 4-16

Create Choropleth Map from Table Data

This example shows how to create a choropleth map from spatial and attribute data contained in tables. A *choropleth map* indicates the values of numeric attributes within spatial regions by using colors along a gradient. Choropleth maps are commonly used to illustrate population density within specified regions.

To create a classification map, where discrete colors indicate the values of classified or categorical attributes, see “Create Classification Map from Table Data” on page 4-92.

Read Data into Tables

Read a comma-separated text file into a table. The file contains farmland statistics for each US state and the District of Columbia, based on the National Agricultural Statistics Service (NASS) Farms and Land in Farms 2019 Summary. For more information about the statistics, see [1] on page 4-91. Display the first row of the table.

```
farms = readtable("farmland.txt",TextType="string");
farms(1,:)
```

```
ans=1×4 table
    StateName    NumberOfFarms    AcresInFarms    AverageFarmSize
    _________    _____________    ____________    _______________

    "Alabama"        38800            8300              214
```

To create a map from the table, you also need spatial information. Read a shapefile containing polygon shapes for each of the US states and the District of Columbia into a geospatial table. A *geospatial table* has a `Shape` variable containing shape information for each row. Display the first row of the geospatial table.

```
states = readgeotable("usastatelo.shp");
states(1,:)
```

```
ans=1×5 table
       Shape          Name       LabelLat    LabelLon    PopDens2000
    ____________    _________    ________    ________    ___________

    geopolyshape    "Alabama"     32.283     -86.921        87.6
```

The `Shape` variable of the table contains `geopolyshape` objects, which are polygons in geographic coordinates. Display the properties of the `geopolyshape` object in the first row.

```
states.Shape(1)
```

```
ans =
  geopolyshape with properties:

              NumRegions: 1
                NumHoles: 0
                Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Join Tables

The table of farmland statistics identifies the District of Columbia as Washington DC, which does not match the corresponding row of the table of polygon shapes. Change the name in the table of farmland statistics so that it matches the name in the table of polygon shapes.

```
dcRow = farms.StateName == "Washington DC";
farms.StateName(dcRow) = "District of Columbia";
```

Join the tables, using the state name variables `Name` and `StateName` as the key variables.

```
statesFarms = outerjoin(states,farms,LeftKey="Name",RightKey="StateName");
```

For more information about how to combine vector data using table joins, see “Combine Vector Data Using Table Joins” on page 6-14.

Create Choropleth Map

Create a choropleth map that shows the average farm size for each state in the conterminous US.

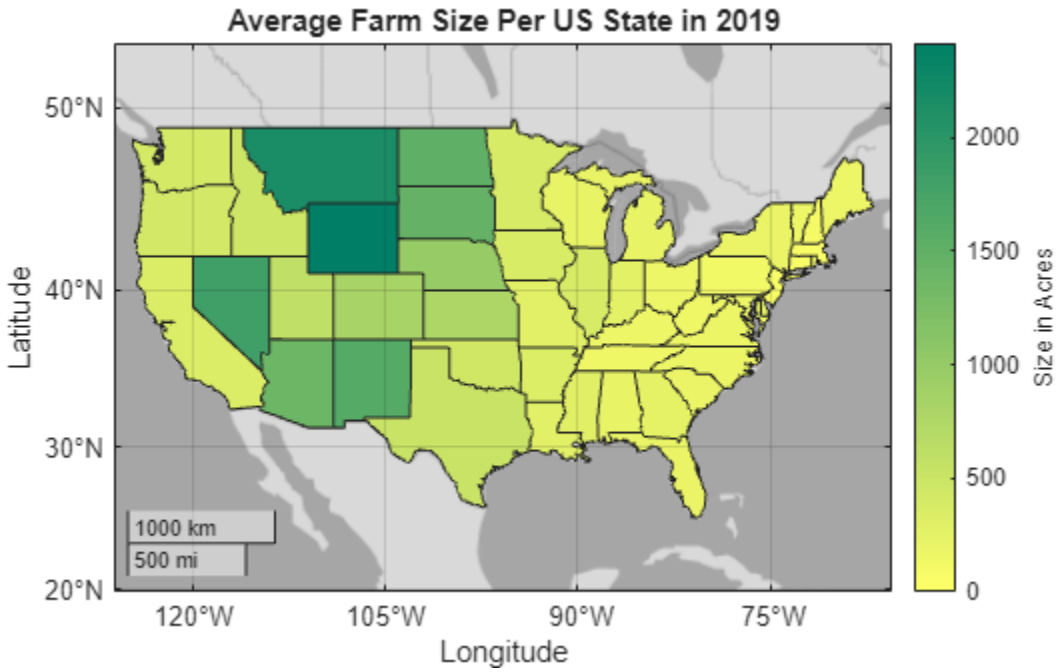
```
figure
rows = statesFarms.Name ~= "Hawaii" & statesFarms.Name ~= "Alaska";
statesFarmsConus = statesFarms(rows,:);
geoplot(statesFarmsConus,ColorVariable="AverageFarmSize");
```

Change the basemap. Add a title and a labeled colorbar.

```
geobasemap darkwater
title("Average Farm Size Per US State in 2019")
cb = colorbar;
cb.Label.String = "Size in Acres";
```

Change the colormap so that, as the average farm sizes increase, the colors of the states transition from yellow to green.

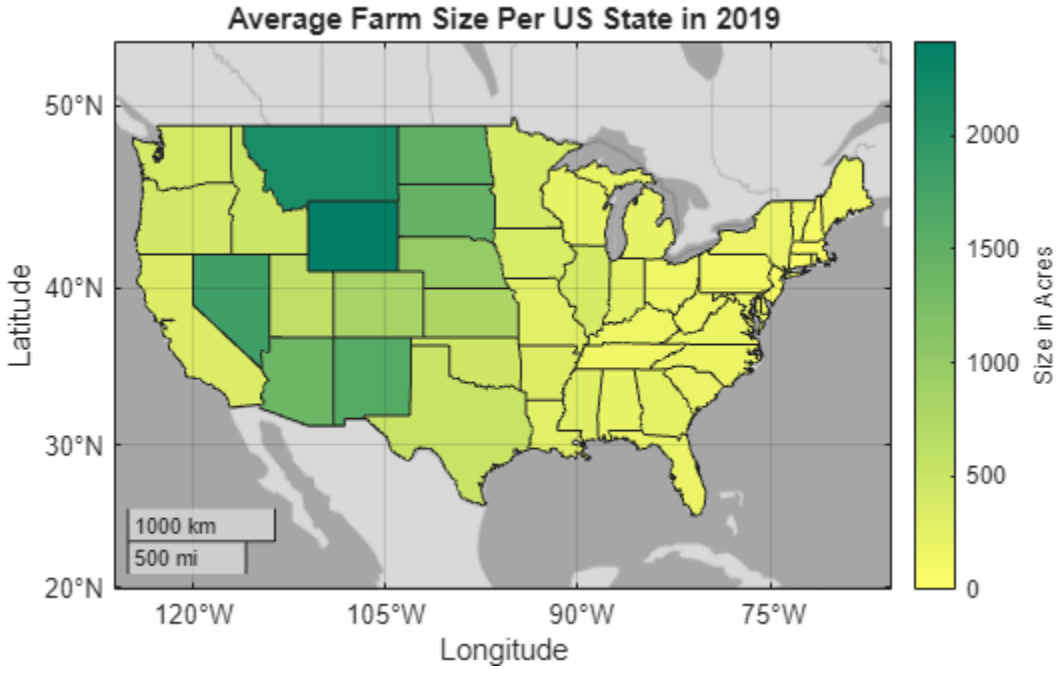
```
cmap = flip(summer);
colormap(cmap)
```



Export Map

Export the map to a PNG file.

```
gx = gca;
exportgraphics(gx,"AverageFarmSizeForUS.png")
```



References

[1] National Agricultural Statistics Service. “Number of Farms, Land in Farms, and Average Farm Size — States and United States: 2018–2019.” *In Farms and Land in Farms 2019 Summary*, 6. USDA, February 2020. https://www.nass.usda.gov/Publications/Todays_Reports/reports/fnlo0220.pdf.

See Also

Functions

`readgeotable` | `geoplot` | `colormap`

Properties

Point Properties | Line Properties | Polygon Properties

Related Examples

- “Create Geospatial Tables” on page 6-2
- “Create Classification Map from Table Data” on page 4-92

Create Classification Map from Table Data

The example shows how to create a classification map from spatial and attribute data contained in tables. A *classification map* indicates the values of classified or categorical attributes within spatial regions by using discrete colors. Classification maps are commonly used to illustrate climate classes, land cover classes, and binned population density within specified regions.

To create a choropleth map, where colors along a gradient indicate the values of numeric attributes, see “Create Choropleth Map from Table Data” on page 4-88.

Read Data into Tables

Read a comma-separated text file into a table. The file contains farmland statistics for each US state and the District of Columbia, based on the National Agricultural Statistics Service (NASS) Farms and Land in Farms 2019 Summary. For more information about the statistics, see [1] on page 4-95. Display the first row of the table.

```
farms = readtable("farmland.txt","TextType","string");
farms(1,:)
```

```
ans=1×4 table
    StateName    NumberOfFarms    AcresInFarms    AverageFarmSize
    _________    _____________    ____________    _______________

    "Alabama"        38800            8300              214
```

To create a map from the table, you also need spatial information. Read a shapefile containing polygon shapes for each of the US states and the District of Columbia into a geospatial table. A *geospatial table* has a `Shape` variable containing shape information for each row. Display the first row of the geospatial table.

```
states = readgeotable("usastatelo.shp");
states(1,:)
```

```
ans=1×5 table
       Shape          Name       LabelLat    LabelLon    PopDens2000
    ____________    _________    ________    ________    ___________

    geopolyshape    "Alabama"     32.283     -86.921        87.6
```

The `Shape` variable of the table contains `geopolyshape` objects, which are polygons in geographic coordinates. Display the properties of the `geopolyshape` object in the first row.

```
states.Shape(1)
```

```
ans =
  geopolyshape with properties:

              NumRegions: 1
                NumHoles: 0
                Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Join Tables

The table of farmland statistics identifies the District of Columbia as Washington DC, which does not match the corresponding row of the table of polygon shapes. Change the name in the table of farmland statistics so that it matches the name in the table of polygon shapes.

```
dcRow = farms.StateName == "Washington DC";
farms.StateName(dcRow) = "District of Columbia";
```

Join the tables, using the state name variables `Name` and `StateName` as the key variables.

```
statesFarms = outerjoin(states,farms,"LeftKey","Name","RightKey","StateName");
```

For more information about how to combine vector data using table joins, see “Combine Vector Data Using Table Joins” on page 6-14.

Discretize Data

Group the average farm size for each state into 4 bins of type `categorical`. View the unique bins.

```
n = 4;
[BinnedAverageFarmSize,E] = discretize(statesFarms.AverageFarmSize,n,"categorical");
bins = unique(BinnedAverageFarmSize)
```

```
bins = 4×1 categorical
     [0, 700)
     [700, 1400)
     [1400, 2100)
     [2100, 2800]
```

Add the binned data to the geospatial table.

```
statesFarms = addvars(statesFarms,BinnedAverageFarmSize);
```

Create Map

Display the binned average farm size for states in the conterminous US by creating a classification map with a legend.

Set up a topographic map.

```
figure
geobasemap topographic
hold on
```

To include each bin in the legend as a separate data series, you must separately plot the states within each bin. For each bin:

- Find the indices of the table rows that are associated with the bin.
- Extract the rows into a subtable.
- Plot the data within the subtable, using the bin as the display name.

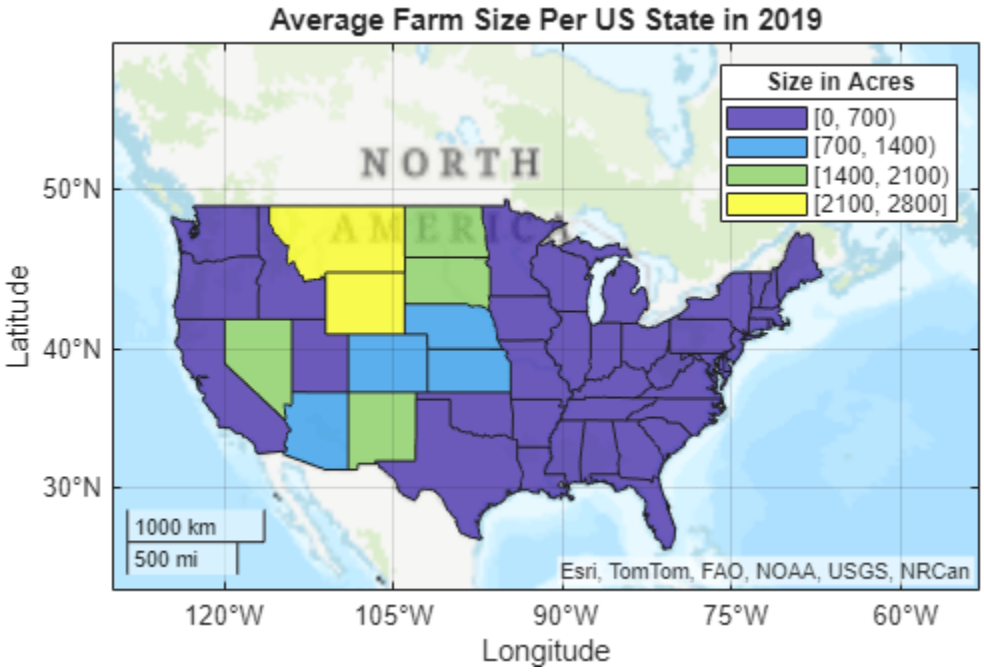
```
for k = 1:n
    idx = statesFarms.BinnedAverageFarmSize == bins(k);
    subT = statesFarms(idx,:);
    geoplot(subT,DisplayName=string(bins(k)))
end
```

Update the geographic limits to include the region surrounding the conterminous US. Add a title and legend.

```
geolimits([22 57.5],[-130 -53])
title("Average Farm Size Per US State in 2019")
lgd = legend;
title(lgd,"Size in Acres")
```

Change the colormap and increase the opacity of the states.

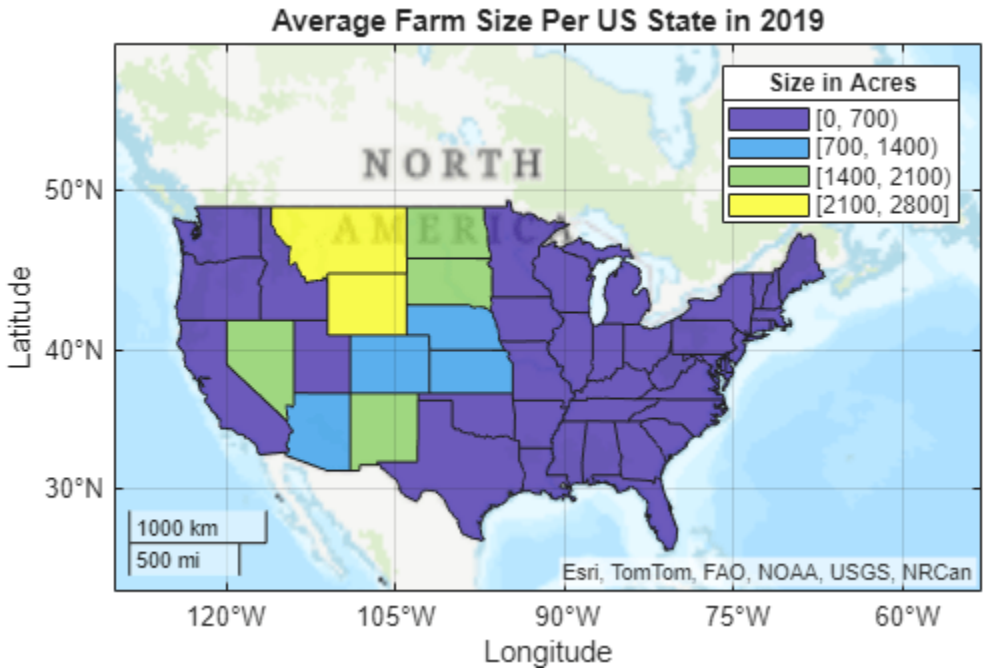
```
gx = gca;
gx.ColorOrder = parula(n);
alpha(0.75)
```



Export Map

Export the map to a PNG file.

```
gx = gca;
exportgraphics(gx,"AverageFarmSizeClassification.png")
```



References

[1] National Agricultural Statistics Service. “Number of Farms, Land in Farms, and Average Farm Size — States and United States: 2018–2019.” *In Farms and Land in Farms 2019 Summary*, 6. USDA, February 2020. https://www.nass.usda.gov/Publications/Todays_Reports/reports/fnlo0220.pdf.

See Also

Functions
`readgeotable` | `geoplot` | `colormap`

Properties
Point Properties | Line Properties | Polygon Properties

Related Examples

- “Create Geospatial Tables” on page 6-2
- “Create Choropleth Map from Table Data” on page 4-88

3-D Visualization

- “Access Basemaps and Terrain for Geographic Globe” on page 5-2
- “Create Interactive Basemap Picker” on page 5-4
- “Visualize Aircraft Line-of-Sight over Terrain” on page 5-6
- “Visualize UAV Flight Path on 2-D and 3-D Maps” on page 5-15
- “Add Light Source to Terrain Map” on page 5-25
- “Surface Relief Shading” on page 5-28
- “Colored Surface Shaded Relief” on page 5-32
- “Relief Mapping with Light Objects” on page 5-35
- “Drape Data on Elevation Maps” on page 5-39
- “Drape Geoid Heights over Topography” on page 5-44

Access Basemaps and Terrain for Geographic Globe

Geographic globe objects created using the `geoglobe` function plot data over basemaps and terrain. You can access different basemap and terrain choices in different ways.

MathWorks offers a selection of basemaps, including two-tone, color terrain, and high-zoom-level displays. Six of the basemaps are tiled data sets created using Natural Earth. Five of the basemaps are high-zoom-level maps provided by Esri. For more information about basemap options, see the `geobasemap` function.

Use Installed Basemap

The `"darkwater"` basemap is installed with MATLAB. The other basemaps are not installed with MATLAB, but you can access them over an internet connection.

Download Basemaps

To work offline or to improve map responsiveness, you can download the basemaps created using Natural Earth onto your local system. The basemaps provided by Esri are not available for download.

Download basemaps using the Add-On Explorer.

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Add-Ons**.
2. In the Add-On Explorer, scroll to the **MathWorks Optional Features** section, and click **Show All** to find the basemap add-ons. You can also search for the basemap add-ons by name (listed in the following table) or click **Optional Features** in **Filter by Type**.
3. Select the basemap add-ons that you want to download.

Basemap Name	Basemap Data Package Name
`'bluegreen'`	MATLAB Basemap Data - bluegreen
`'grayland'`	MATLAB Basemap Data - grayland
`'colorterrain'`	MATLAB Basemap Data - colorterrain
`'grayterrain'`	MATLAB Basemap Data - grayterrain
`'landcover'`	MATLAB Basemap Data - landcover

Add Custom Basemaps

Add a custom basemap from a URL, an MBTiles file, a georeferenced image, or a georeferenced data grid by using the `addCustomBasemap` function. MATLAB requires an active internet connection to add and use custom basemaps from a URL. MATLAB does not require internet access to add custom basemaps from MBTiles files, images, or data grids.

MBTiles files are useful for storing and sharing tiled basemap data. You can write a georeferenced image or data grid to an MBTiles file by using the `mbtileswrite` function.

Access Terrain

By default, the geographic globe uses terrain data hosted by MathWorks and derived from the GMTED2010 model by the USGS and NGA. You need an active internet connection to access this terrain data, and you cannot download it.

To work offline or to improve terrain responsiveness, add custom terrain from DTED files using the `addCustomTerrain` function. You do not need an active internet connection to add or use custom terrain.

Alternatively, you can set the `Terrain` property of the geographic globe object to `'none'`.

Specify Basemaps and Terrain

To specify a basemap for a geographic globe, you can either:

- Use the `geobasemap` function. Specify the geographic globe as the first argument.

  ```
  uif = uifigure;
  g = geoglobe(uif);
  geobasemap(g,'streets')
  ```

- Set the `Basemap` property of the `GeographicGlobe` object. You can set this property by using a name-value pair or by using dot notation.

  ```
  uif = uifigure;
  g = geoglobe(uif,'Basemap','streets');
  g.Basemap = 'topographic';
  ```

To specify terrain for the geographic globe, set the `Terrain` property of the `GeographicGlobe` object. You can set this property by using a name-value pair or by using dot notation.

```
uif = uifigure;
g = geoglobe(uif,'Terrain','none');
g.Terrain = 'gmted2010';
```

See Also

`geoglobe` | `geobasemap` | `addCustomTerrain` | `addCustomBasemap` | `mbtileswrite`

More About

- “Use Basemaps in Offline Environments” on page 4-64
- “System Requirements for Graphics”

Create Interactive Basemap Picker

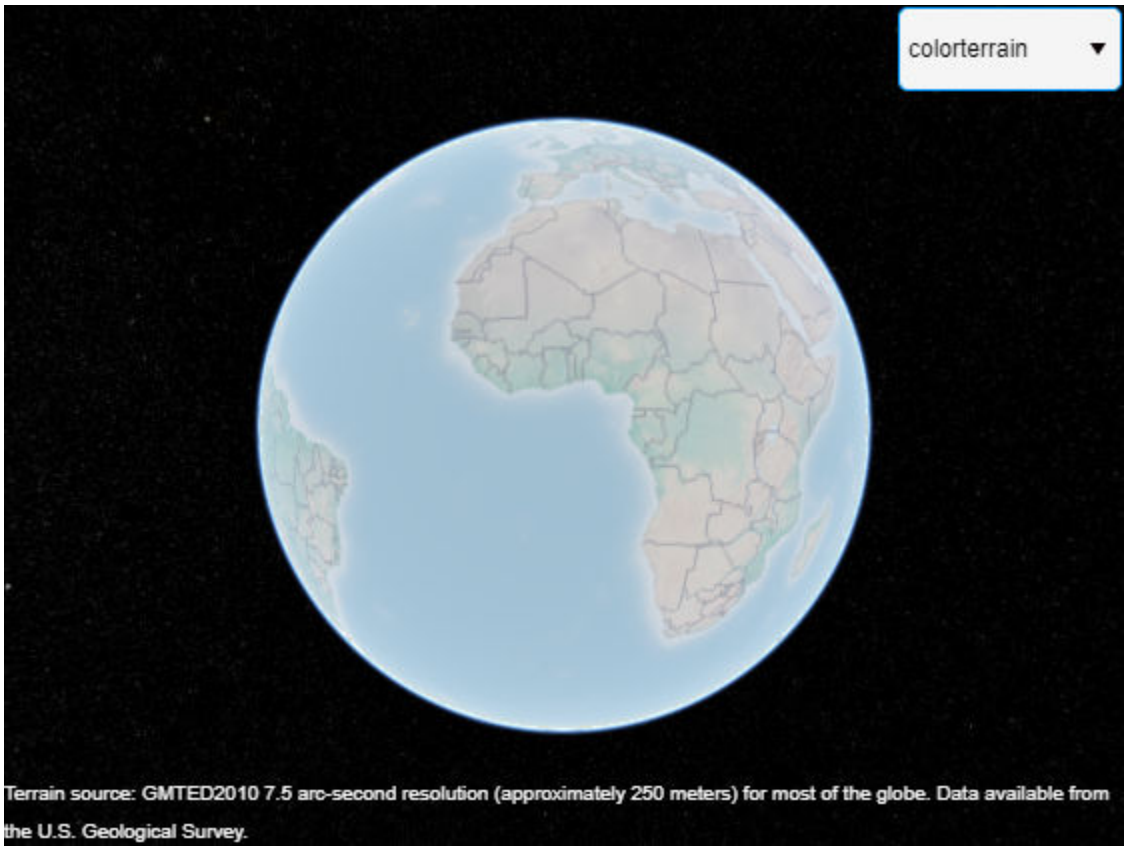
Interactively change the basemap of a geographic globe by adding a drop-down menu to the figure.

First, create a program file called `basemapPicker.m`. Within the program file:

- Create a geographic globe in a figure created using the `uifigure` function.
- Specify a position for the menu. In this example, the values of `x`, `y`, `w`, and `h` position the menu in the upper-right corner of the figure window.
- Specify the basemaps to include in the menu.
- Create the menu. Use a `ValueChangedFcn` callback that executes when you make a selection from the menu. The callback changes the basemap using the `geobasemap` function.
- Write custom code to reposition the menu when you change the size of the figure. To do this, disable automatic resizing of the menu. Then, create custom behavior by defining a `SizeChangedFcn` callback. The `repositionDropdown` function repositions the menu, so that it stays in the upper-right corner of the figure.

```
function basemapPicker
    uif = uifigure;
    gl = geoglobe(uif);

    x = 0.8;
    y = 0.9;
    w = 0.2;
    h = 0.1;
    uifW = uif.Position(3);
    uifH = uif.Position(4);
    pos = [x*uifW y*uifH w*uifW h*uifH];

    basemaps = ["satellite" "streets" "streets-light" "streets-dark" ...
                "landcover" "darkwater" "grayland" "bluegreen" ...
                "grayterrain" "colorterrain"];

    dd = uidropdown(uif,'Position',pos,'Items',basemaps);
    dd.ValueChangedFcn = @(src,eventdata)geobasemap(gl,src.Value);

    uif.AutoResizeChildren = 'off';
    uif.SizeChangedFcn = @(src,eventdata)repositionDropdown(dd,x,y,w,h);
end

function repositionDropdown(dd,x,y,w,h)
    fig = dd.Parent;
    uifW = fig.Position(3);
    uifH = fig.Position(4);
    dd.Position = [x*uifW y*uifH w*uifW h*uifH];
end
```

Run the program file. Change the basemap to `'colorterrain'` using the drop-down menu.



See Also

`geoglobe` | `geobasemap` | `uidropdown`

More About

- “Create Callbacks for Graphics Objects”

Visualize Aircraft Line-of-Sight over Terrain

This example shows how to compute and visualize the line-of-sight visibility of an aircraft from a ground location over terrain. First, import terrain data for a region and apply it to a 3-D geographic globe. Then, perform point-to-point visibility analysis from a ground location to a simulated flight path, and display the results on a 3-D geographic globe. Finally, perform point-to-area visibility analysis from the ground location corresponding to aircraft flying at constant altitude, and display the results on a 2-D geographic axes.

Use line-of-sight analysis for ground-to-air scenarios where unobstructed visibility is important, such as for radar surveillance, communications, and UAV path planning. This example applies the analysis to radar surveillance for an airport.

Import Terrain Data

Specify a DTED-format terrain file to use for data analysis and 3-D visualization. The terrain file was downloaded from the "SRTM Void Filled" data set available from the United States Geological Survey (USGS).

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.";
```

Import DTED file data into the workspace as an array and a geographic raster reference object, specifying the return type as double so that the data works with all analysis functions.

```
[Zterrain,Rterrain] = readgeoraster(dtedfile,OutputType="double");
```

View the geographic limits and sample resolution of the terrain data by accessing properties of the geographic raster reference object. The limits for the file correspond to the region around Boulder, Colorado, US, and the resolution corresponds to the DTED level-1 format, which has sample resolution of 3 arc seconds, or about 90 meters.

```
latlim = Rterrain.LatitudeLimits;
lonlim = Rterrain.LongitudeLimits;
latspc = Rterrain.SampleSpacingInLatitude;
lonspc = Rterrain.SampleSpacingInLongitude;
disp("Latitude limits of terrain: " + mat2str(latlim) + newline + ...
    "Longitude limits of terrain: " + mat2str(lonlim) + newline + ...
    "Terrain resolution in latitude: " + latspc*3600 + " arc seconds" + newline + ...
    "Terrain resolution in longitude: " + lonspc*3600 + " arc seconds")
```

```
Latitude limits of terrain: [39 40]
Longitude limits of terrain: [-106 -105]
Terrain resolution in latitude: 3 arc seconds
Terrain resolution in longitude: 3 arc seconds
```

Visualize Aircraft Trajectory Line-of-Sight on a 3-D Map

Create Geographic Globe with Custom Terrain

Add custom data with the DTED file for use with 3-D visualization.

```
addCustomTerrain("southboulder",dtedfile,Attribution=attribution)
```

Specify the custom terrain with a new geographic globe.

```
fig = uifigure;
g = geoglobe(fig,Terrain="southboulder");
```

View Radar Ground Location

Define a radar ground location at Rocky Mountain Metropolitan Airport. The radar is mounted on a tower 10 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height, referenced to mean sea level.

```
rdrlat = 39.913756;
rdrlon = -105.118062;
rdrtowerht = 10;
rdralt = 1717 + rdrtowerht;
```

Plot the radar location on the geographic globe.

```
geoplot3(g,rdrlat,rdrlon,rdralt,"co", ...
    LineWidth=6, ...
    MarkerSize=1)
```

Simulate Aircraft Trajectory

Simulate the trajectory of an aircraft circling over the mountains.

Define the center location of a circling aircraft.

```
tlat0 = 39.80384;
tlon0 = -105.49916;
tht0 = 3000;
```

Define trajectory waypoints for the aircraft using east-north-up (ENU) Cartesian coordinates. Specify a curve with a radius of 5 km (5000 m) and a vertical offset of 1 km (1000 m) over 1.5 revolutions.

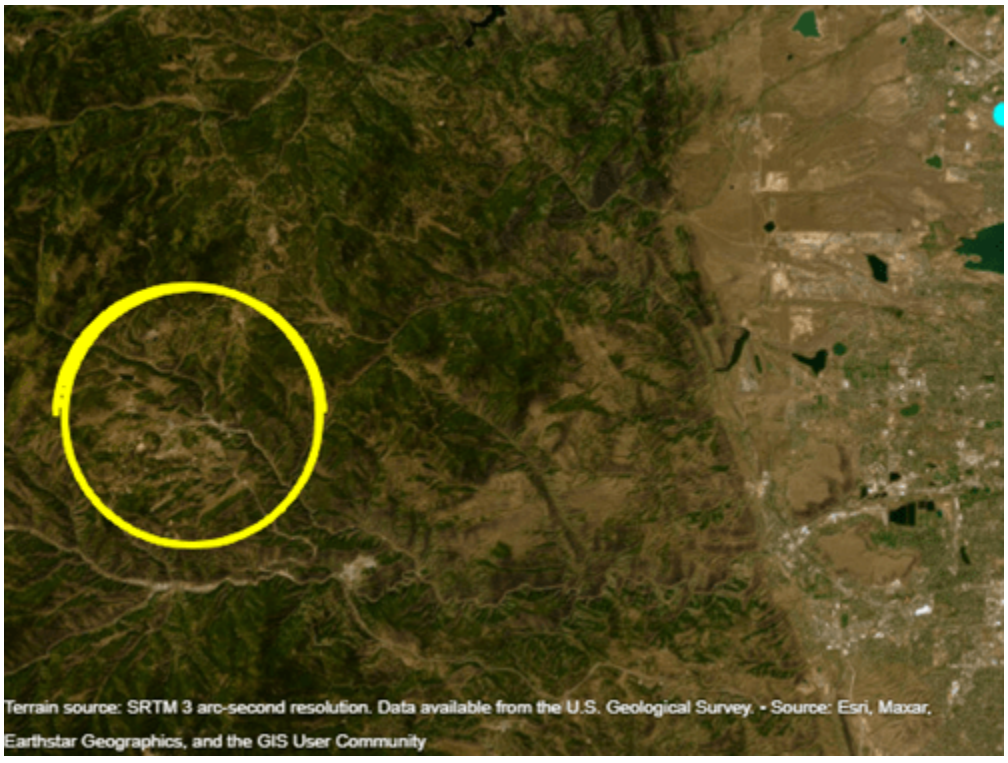
Then, convert the ENU coordinates to geodetic coordinates that are referenced to the WGS84 ellipsoid.

```
azs = 1:540;
r = 5000;
[X,Y] = pol2cart(deg2rad(azs),r);
Z = linspace(0,1000,numel(azs));
wgs84 = wgs84Ellipsoid;
[tlat,tlon,tht] = enu2geodetic(X,Y,Z,tlat0,tlon0,tht0,wgs84);
```

View Aircraft Trajectory over Terrain

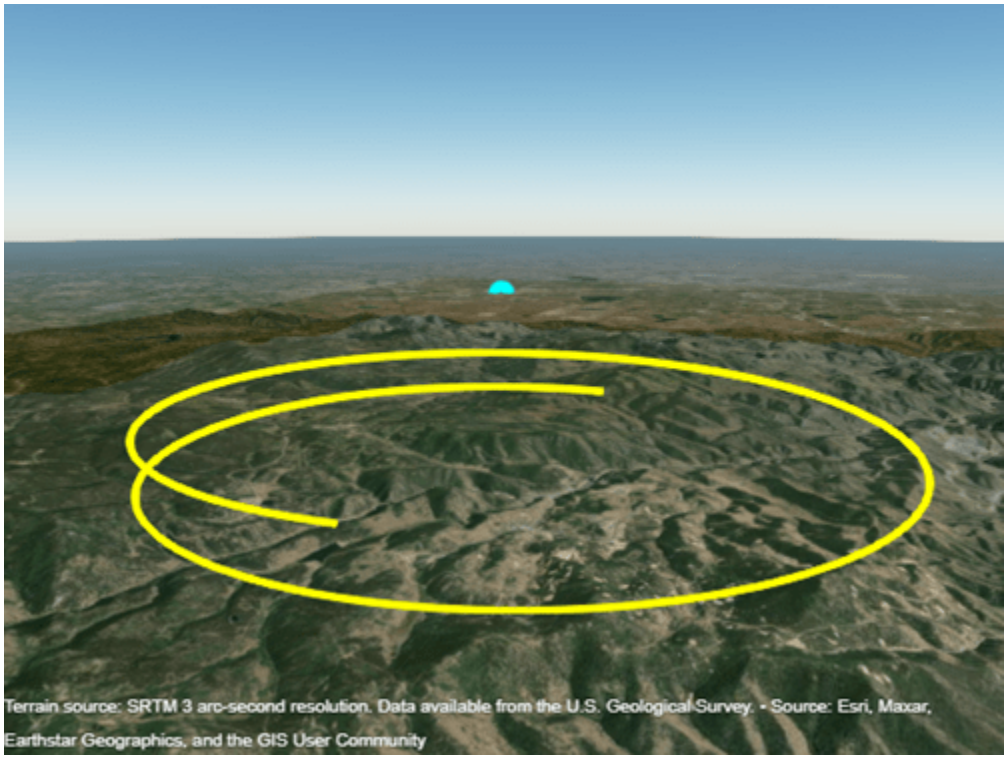
Plot the aircraft trajectory on the geographic globe. The default view, or camera position, is overhead and oriented down.

```
hold(g,"on")
traj = geoplot3(g,tlat,tlon,tht,"y", ...
    HeightReference="ellipsoid", ...
    LineWidth=3);
```



View the 3-D terrain and radar location from a distance by changing the camera position and rotation angles.

```
campos(g,39.77114,-105.62662,6670)
camheading(g,70)
campitch(g,-12)
```



Compute Line-of-Sight Visibility with Aircraft Trajectory

Compute line-of-sight visibility with the `los2` function and the DTED data.

The `los2` function supports either orthometric height (height above mean sea level) or height above ground level. Convert the aircraft trajectory heights from ellipsoidal height to orthometric height. Then, compute the line-of-sight from the airport radar location to each aircraft trajectory waypoint and convert the results to a logical array.

```
numwaypts = numel(tlat);
isvis = zeros(1,numwaypts);
talt = tht - egm96geoid(tlat,tlon);
for k = 1:numwaypts
    isvis(k) = los2(Zterrain,Rterrain,rdrlat,rdrlon,tlat(k),tlon(k),rdralt,talt(k),"MSL","MSL");
end
isvis = logical(isvis);
```

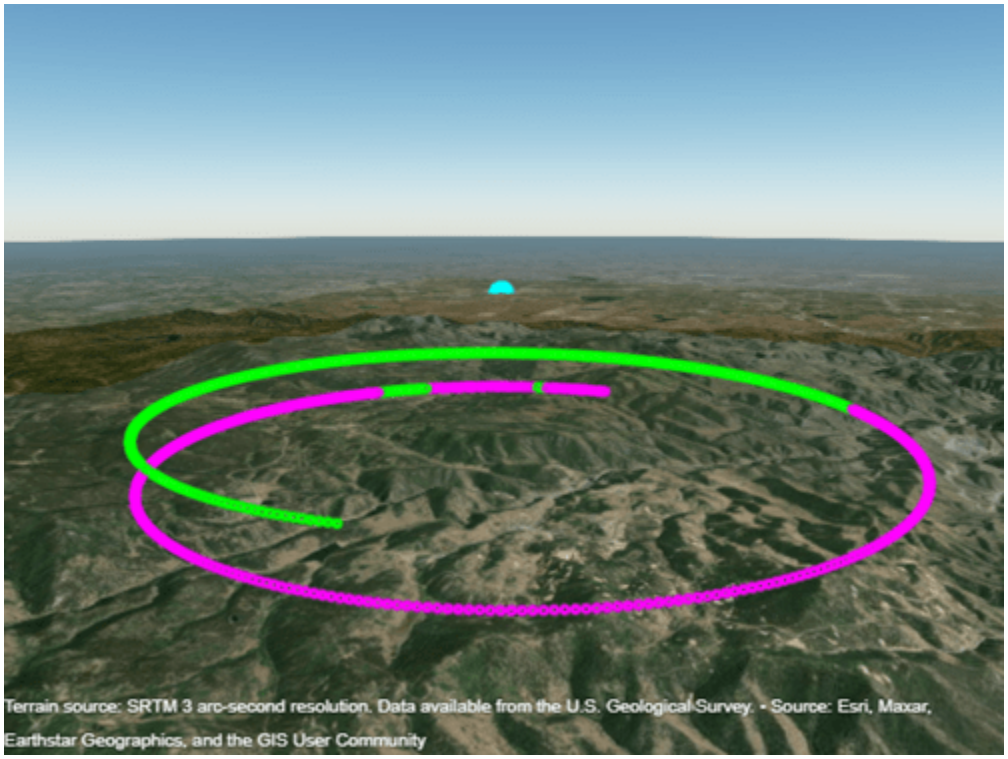
Note that `los2` calculates line-of-sight visibility assuming the data is referenced to a spherical Earth, whereas the data is actually referenced to the WGS84 ellipsoid, and as a result there may be minor discrepancies. The line-of-sight calculation also corresponds to optical line-of-sight and does not account for refraction through the atmosphere.

Visualize Line-of-Sight Visibility over Terrain

Plot the line-of-sight visibility. Use green markers where the aircraft is visible from the airport and magenta markers where it is not visible.

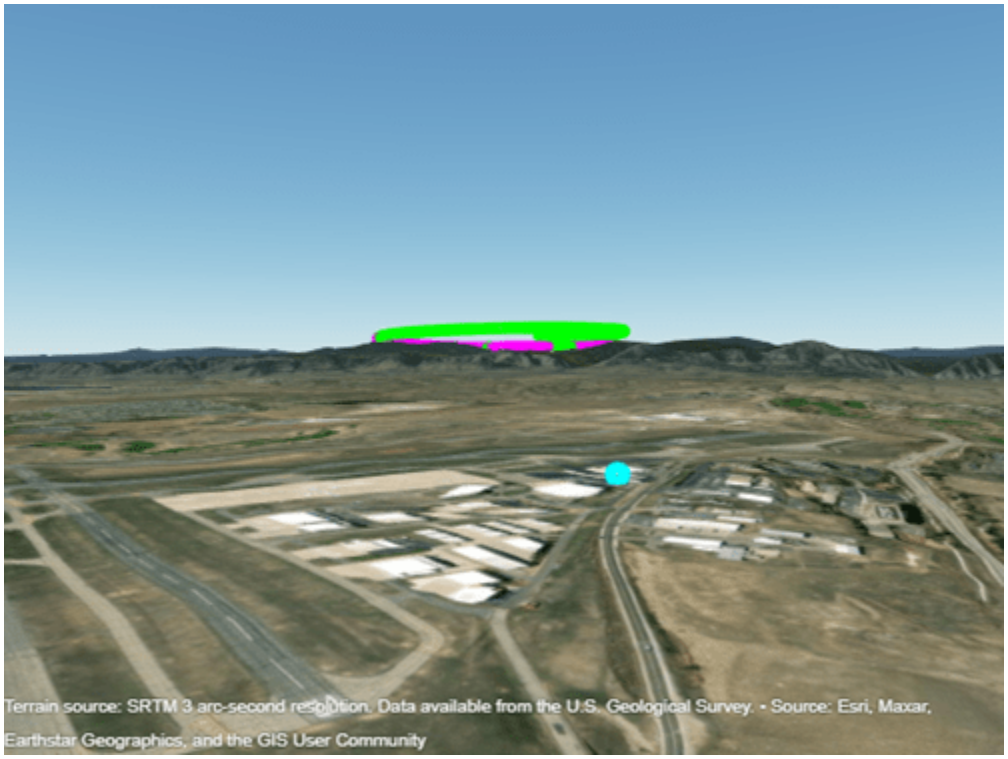
```
delete(traj)
geoplot3(g,tlat(isvis),tlon(isvis),tht(isvis),"og", ...
    HeightReference="ellipsoid", ...
```

```
    LineWidth=2, ...
    MarkerSize=1)
geoplot3(g,tlat(~isvis),tlon(~isvis),tht(~isvis),"om", ...
    HeightReference="ellipsoid", ...
    LineWidth=2, ...
    MarkerSize=1)
```



View the line-of-sight plot from the perspective of the airport. Get the geodetic coordinates of the position that is 900 meters east, 200 meters north, and 100 meters up from the radar location. Then, set the camera position and rotation angles. The green markers appear in view, but the magenta markers are either completely or partially obstructed by terrain.

```
rdrht = rdralt + egm96geoid(rdrlat,rdrlon);
[camlat,camlon,camht] = enu2geodetic(900,200,100,rdrlat,rdrlon,rdrht,wgs84);
campos(g,camlat,camlon,camht)
camheading(g,-110)
campitch(g,0)
```



Visualize Aircraft Line-of-Sight Visibility Contours on a 2-D Map

The previous sections performed point-to-point line-of-sight analysis and visualization from a radar location to an aircraft trajectory. Now perform point-to-area line-of-sight analysis and visualization from the same radar location over the terrain region. The visualization displays the edge of visibility for an aircraft flying at constant altitude.

Plot Radar Location and Terrain Limits on a 2-D Map

Plot the radar location in a new figure with a topographic 2-D map.

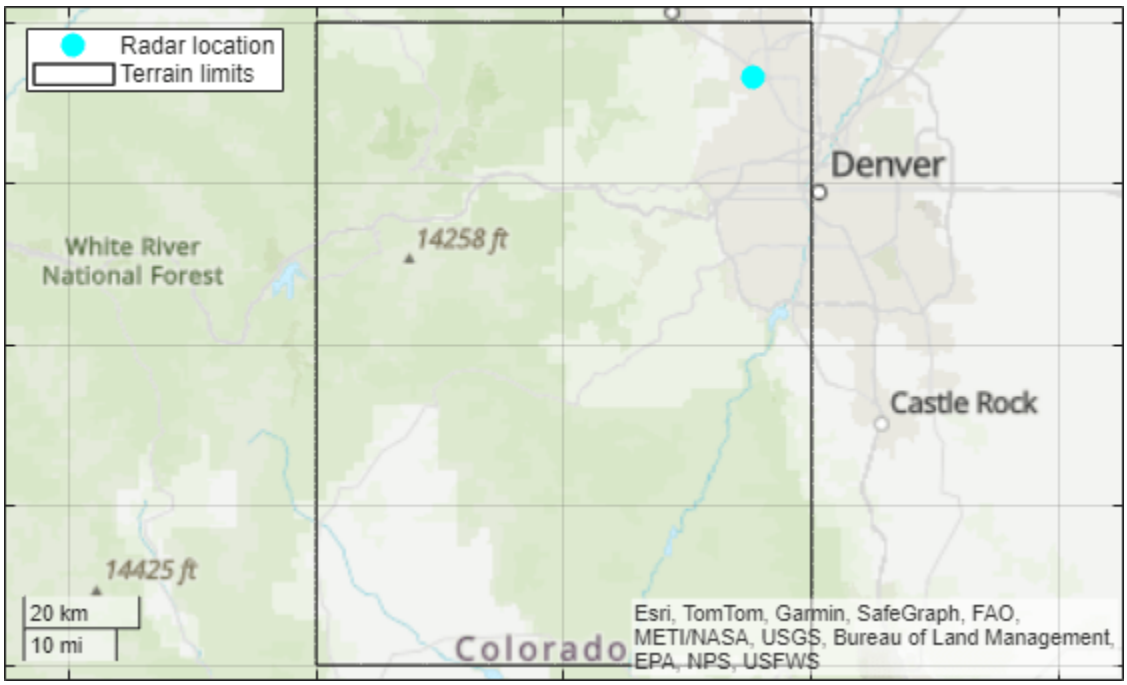
```
figure
geoplot(rdrlat,rdrlon,"co", ...
    LineWidth=6, ...
    MarkerSize=3, ...
    DisplayName="Radar location")
geobasemap topographic
gx = gca;
gx.InnerPosition = gx.OuterPosition;
```

Display the limits of the custom terrain as an area of interest (AOI) on the map.

```
aoi = aoiquad(latlim,lonlim);
hold on
geoplot(aoi, ...
    LineWidth=1, ...
    FaceColor="none", ...
    DisplayName="Terrain limits")
```

Display a legend in the northwest corner.

```
legend(Location="northwest")
```



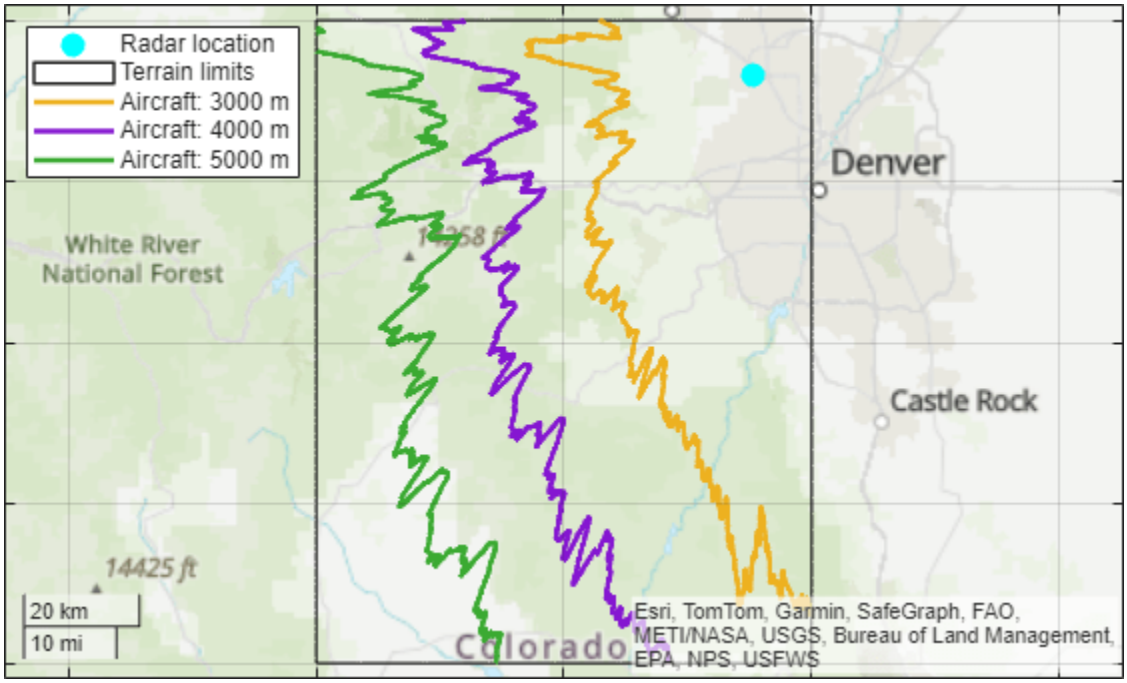
Plot Visibility Contours for Aircraft Flying at Constant Altitude

Specify three altitudes above mean sea level for the aircraft. For each altitude:

- Compute the viewshed using the radar location as the observer. The viewshed defines the area that has line-of-sight visibility.
- Find the edge of aircraft visibility by computing contours from the viewshed data.
- Remove small contour segments.
- Plot the contours on the geographic axes.

```
tgtalts = [3000 4000 5000];

minVertices = 10;
cfig = figure("Visible","off"); % Suppress contour plot using invisible figure
cax = axes("Parent",cfig);
for tgtalt = tgtalts
    vis = viewshed(Zterrain,Rterrain,rdrlat,rdrlon,rdralt,tgtalt,"MSL","MSL");

    C = contourm(vis,Rterrain,"LevelList",1,"Parent",cax);
    clat = C(2,:);
    clon = C(1,:);

    clats = [];
    clons = [];
    k = 1;
    while k < size(C,2)
        numVertices = clat(k);
        if numVertices > minVertices % Do not plot small segments
            clats = [clats clat(k+1:k+numVertices) NaN]; %#ok<AGROW>
            clons = [clons clon(k+1:k+numVertices) NaN]; %#ok<AGROW>
```

```
        end
        k = k + numVertices + 1;
    end

    geoplot(gx,clats,clons,"LineWidth",2, ...
        "DisplayName", "Aircraft: " + string(tgtalt) + " m");
end
```



The contours appear primarily to the west of the radar location over the mountains. The contours do not appear in other directions because visibility is not constrained by the terrain in those directions within the terrain data limits.

If the radar is constrained by line-of-sight visibility, then the contours correspond to radar coverage regions for varying altitude, where the nearest contour to the radar corresponds to radar coverage for an aircraft flying at 3000 meters and the furthest contour corresponds to radar coverage for an aircraft flying at 5000 meters.

As with `los2`, the `viewshed` function calculates line-of-sight visibility assuming the data is referenced to a spherical Earth, whereas the data is actually referenced to the WGS84 ellipsoid, and as a result there may be minor discrepancies. The line-of-sight calculation also corresponds to optical line-of-sight and does not account for refraction through the atmosphere.

Clean Up

Clean up by closing the geographic globe and removing the imported terrain data.

```
if isvalid(fig)
    close(fig)
```

```
end
removeCustomTerrain("southboulder")
```

See Also

Functions
`addCustomTerrain` | `geoplot3` | `geoglobe` | `los2` | `campos` | `readgeoraster`

Objects
`GeographicPostingsReference`

More About

- “Find Ellipsoidal Height from Orthometric Height” on page 8-27
- “Find Geospatial Raster Data” on page 2-9

Visualize UAV Flight Path on 2-D and 3-D Maps

This example visualizes a simulated unmanned aerial vehicle (UAV) flight from the Mauna Loa Baseline Observatory to the top of the Mauna Loa Volcano in Hawaii. First, display the track on geographic axes and a geographic globe. Then, synchronize the view and visualize the flight path by using camera navigation functions. Finally, view the top of the Mauna Loa volcano as a panorama.

Visualize Area of Interest in 2-D

The use of UAVs to track characteristics of changing topology, gasses, and ash clouds around volcanoes is becoming an important area of research for scientists [1]. A UAV can travel in areas of interest (AOIs) that are hazardous for a volcanologist. Simulating the flight path of the UAV prior to sending it out on a mission can assist with understanding the topology and terrain. To gain an overview and 2-D perspective of the AOI, view the locations of the Mauna Loa Baseline Observatory and the Mauna Loa Volcano in a geographic axes.

Get Coordinates of Mauna Loa Baseline Observatory

Specify the coordinates of the Mauna Loa Baseline Observatory [2]. The height of the observatory is in meters above mean sea level (MSL).

```
obslat = 19.5362;
obslon = -155.5763;
obsH = 3397.00;
```

Get Coordinates of Mauna Loa Volcano

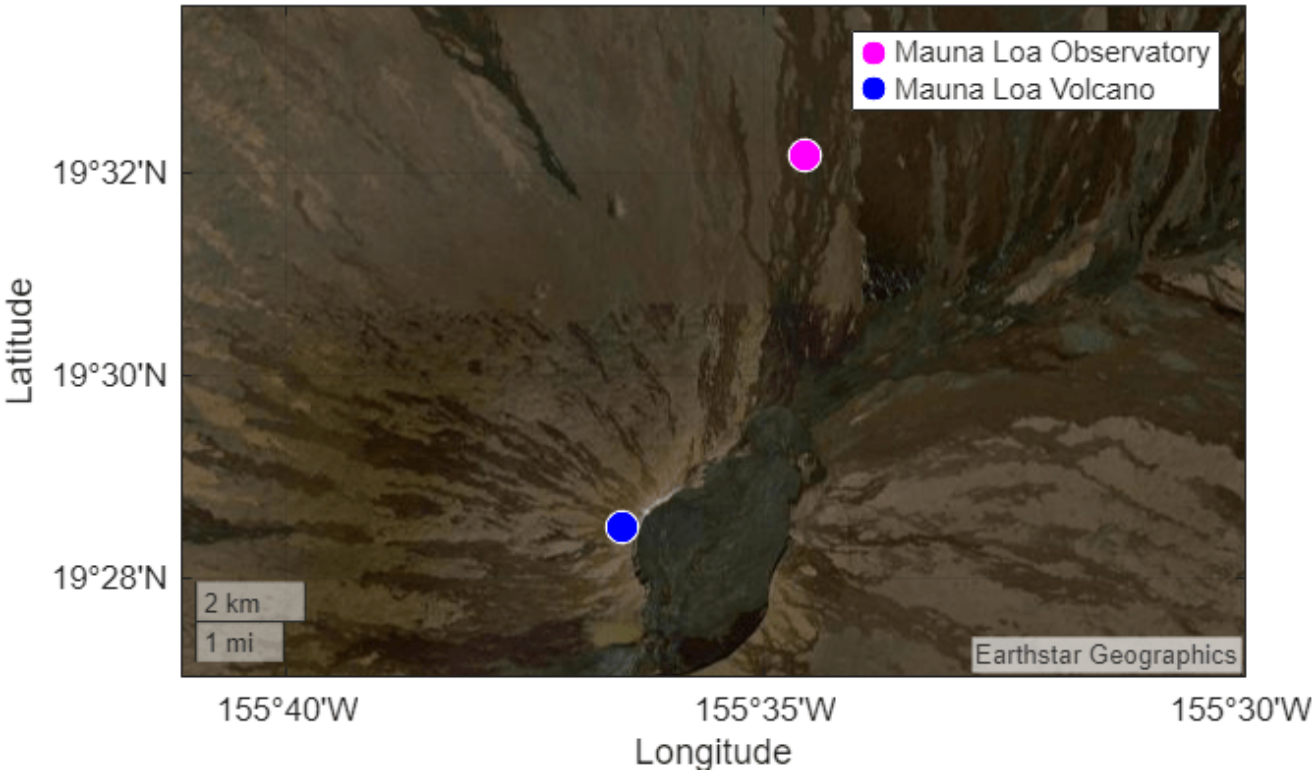
Specify the coordinates of the top of Mauna Loa [3]. The height of the volcano is orthometric and is in meters.

```
mllat = 19.475;
mllon = -155.608;
mlH = 4169;
```

View Mauna Loa Baseline Observatory and Mauna Loa Volcano in 2-D

For a 2-D perspective of the AOI, use `geoaxes` and `geoplot` to plot the location of the observatory and the top of the volcano.

```
figure
geoaxes(Basemap="satellite",ZoomLevel=12)
hold on
geoplot(obslat,obslon,"ow",MarkerSize=10,MarkerFaceColor="magenta",DisplayName="Mauna Loa Observa
geoplot(mllat,mllon,"ow",MarkerSize=10,MarkerFaceColor="blue",DisplayName="Mauna Loa Volcano")
legend
```



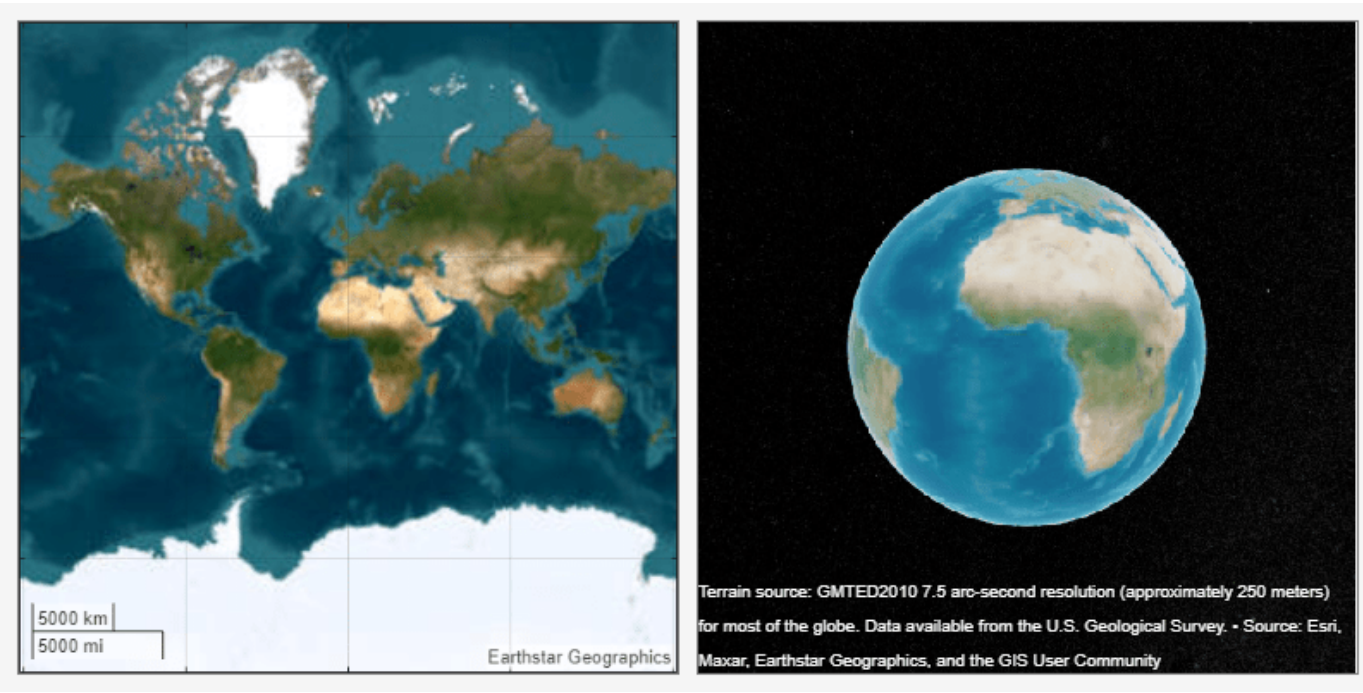
Synchronize View of Mauna Loa Baseline Observatory in 2-D and 3-D

Use the geographic axes to view the observatory in 2-D and the geographic globe to view the observatory in 3-D.

Create Geographic Axes and Geographic Globe in Same Figure

Set up a 2-D and 3-D map display by creating geographic axes and a geographic globe in the same UI figure. To view more of the 2-D map, set the `InnerPosition` of the geographic axes to its `OuterPosition`. To view both map displays with the same basemap, set the basemap of the geographic axes to `"satellite"`.

```
figpos = [1000 500 800 400];
uif = uifigure(Position=figpos);
ug = uigridlayout(uif,[1,2]);
p1 = uipanel(ug);
p2 = uipanel(ug);
gx = geoaxes(p1,Basemap="satellite");
gg = geoglobe(p2);
gx.InnerPosition = gx.OuterPosition;
gg.Position = [0 0 1 1];
```



View Observatory in 2-D

View the observatory from 200 meters above the terrain. Control the view of the geographic axes by changing its map center and zoom level. You can synchronize the view of the geographic axes with the view of the geographic globe by converting the camera height of the globe to a zoom level for the axes. Calculate an approximate zoom level from terrain height by using the `heightToZoomLevel` local function.

```
heightAboveTerrain = 200;
gx.MapCenter = [obslat obslon];
zoomLevel = heightToZoomLevel(heightAboveTerrain,obslat);
gx.ZoomLevel = zoomLevel;
```

View Observatory in 3-D

Control the view of the geographic globe by changing the position of the camera. The `campos` function requires you to specify ellipsoidal height (relative to the WGS84 ellipsoid) instead of orthometric height (relative to mean sea level). Convert the height of the observatory to ellipsoidal height. All heights are in meters.

```
N = egm96geoid(obslat,obslon);
obsh = obsH + N;
ellipsoidalHeight = obsh + heightAboveTerrain;
campos(gg,obslat,obslon,ellipsoidalHeight)
drawnow
```

Import Flight Track Data and Calculate Heading and 3-D Distance

Import the simulated flight track from the Mauna Loa Baseline Observatory to the top of the Mauna Loa volcano. The file contains the latitudes, longitudes, and altitudes of the UAV path, referenced to mean sea level.

```
T = readgeotable("sample_uavtrack.gpx",Layer="track_points");
tlat = T.Shape.Latitude';
tlon = T.Shape.Longitude';
talt = T.Elevation';
```

Calculate Flight Headings

Calculate the UAV heading at each track point using the `azimuth` function. The `azimuth` function calculates values using a clockwise-positive convention.

```
wgs84 = wgs84Ellipsoid;
theading = azimuth(tlat(1:end-1),tlon(1:end-1),tlat(2:end),tlon(2:end),wgs84);
theading = [theading(1);theading(:)];
```

Calculate 3-D Distances

Calculate the cumulative distance for the UAV flight track. The `distance` function does not take into account changes in elevation or altitude. In order to calculate the distance the UAV moves from point to point in 3-D, you need to work in geocentric Cartesian coordinates (X, Y, Z). Compute the point-to-point offset components (in meters) using the `ecefOffset` function. The altitude data of the UAV flight is referenced to the mean sea level. To use the `ecefOffset` function, the heights must be referenced to the ellipsoid. Convert the orthometric heights of the flight track to ellipsoidal height (relative to the WGS84 ellipsoid). All heights are in meters.

```
N = egm96geoid(tlat,tlon);
h = talt + N;
```

Compute distance offsets.

```
lat1 = tlat(1:end-1);
lat2 = tlat(2:end);
lon1 = tlon(1:end-1);
lon2 = tlon(2:end);
h1 = h(1:end-1);
h2 = h(2:end);
[dx,dy,dz] = ecefOffset(wgs84,lat1,lon1,h1,lat2,lon2,h2);
```

Calculate the Euclidean distance between each pair of adjacent points using the `hypot` function. The distance is in meters.

```
distanceIncrementIn3D = hypot(hypot(dx, dy), dz);
```

Compute cumulative distance in 3-D and the total distance in meters.

```
cumulativeDistanceIn3D = cumsum(distanceIncrementIn3D);
totalDistanceIn3D = sum(distanceIncrementIn3D);
fprintf("Total UAV track distance is %f meters.\n",totalDistanceIn3D)
```

```
Total UAV track distance is 8931.072120 meters.
```

Assign a variable for the cumulative distance to be used for plotting the animation.

```
tdist = [0 cumulativeDistanceIn3D];
```

Plot Flight Line from Mauna Loa Baseline Observatory to top of Mauna Loa Volcano

Plot the simulated flight line from the Mauna Loa Baseline Observatory to the top of the Mauna Loa volcano.

Plot the flight line. By default, the geographic globe places the line at the center of the display. The view of the geographic axes will not change because you have previously set the `MapCenter` and `ZoomLevel`.

```
geoplot3(gg,tlat,tlon,talt,"c",LineWidth=2,HeightReference="geoid")
ptrack = geoplot(gx,tlat,tlon,"c",LineWidth=2);
```

Set the map center and zoom level to be consistent with the 3-D view by converting the camera height for the globe to the zoom level for the axes.

```
[clat,clon,cheight] = campos(gg);
gx.MapCenter = [clat clon];
gx.ZoomLevel = heightToZoomLevel(cheight,clat);
drawnow
```

Set Initial View from Mauna Loa Baseline Observatory to Top of Mauna Loa Volcano

View the flight line from the start position by setting the camera position to the first coordinate of the track. For a better perspective, set the camera height to 75 meters about the height of the track. View straight down to the observatory by setting the camera pitch to -90. View the track by setting the heading to the third element of the calculated heading array since the first two points of the track are the same location and the calculated heading for those locations is 0.

```
campos(gg,tlat(1),tlon(1))
camheight(gg,talt(1) + 75)
campitch(gg,-90)
camheading(gg,theading(3))
```

On the 2-D map, show the start and end locations of the flight track using markers, and show the location of the UAV using an icon. Create a legend for the UAV track, the markers, and the icon.

```
hold(gx,"on")
mstart = geoplot(gx,tlat(1),tlon(1),"ow",MarkerSize=10,MarkerFaceColor="magenta");
mend = geoplot(gx,tlat(end),tlon(end),"ow",MarkerSize=10,MarkerFaceColor="blue");
icon = geoiconchart(gx,tlat(1),tlon(1),"uav.png",SizeData=30);

icon.DisplayName = "Current Location";
mstart.DisplayName = "Start Location";
mend.DisplayName = "End Location";
ptrack.DisplayName = "UAV Track";
legend(gx)
```

Add the USGS Topographic Map basemap from the USGS National Map. Then, view the topology of the AOI by changing the basemap and zoom level.

```
url = "https://basemap.nationalmap.gov/ArcGIS/rest/services/USGSTopo/MapServer/tile/${z}/${y}/${x
addCustomBasemap("usgstopo",url,Attribution="USGS The National Map")
```

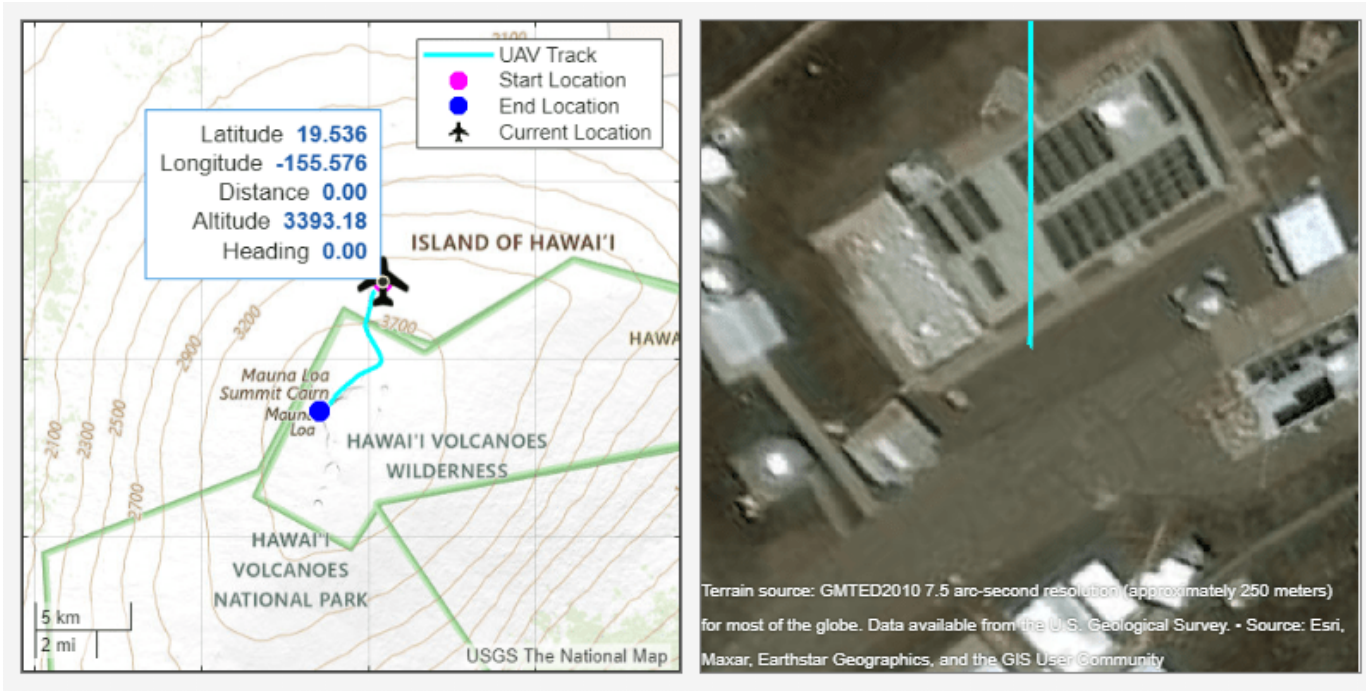
```
gx.Basemap = "usgstopo";
gx.ZoomLevel = 11;
```

Create a data tip that corresponds to the current latitude and longitude of the UAV.

```
dt = datatip(ptrack,DataIndex=1,Location="northwest");
```

Customize the data tip by formatting the latitude and longitude rows, and by adding rows that indicate the altitude of the UAV, the heading of the UAV, and the distance from the observatory.

```
ptrack.DataTipTemplate.DataTipRows(1).Format = "%.3f";
ptrack.DataTipTemplate.DataTipRows(2).Format = "%.3f";

dtrow = dataTipTextRow("Distance",tdist,"%.2f");
dtrow(end+1) = dataTipTextRow("Altitude",talt,"%.2f");
dtrow(end+1) = dataTipTextRow("Heading",theading,"%.2f");
ptrack.DataTipTemplate.DataTipRows(end+1:end+3) = dtrow;
```



Fly from Mauna Loa Baseline Observatory to Top of Mauna Loa Volcano

Animate a flight from the Mauna Loa Baseline Observatory to the top of the Mauna Loa volcano.

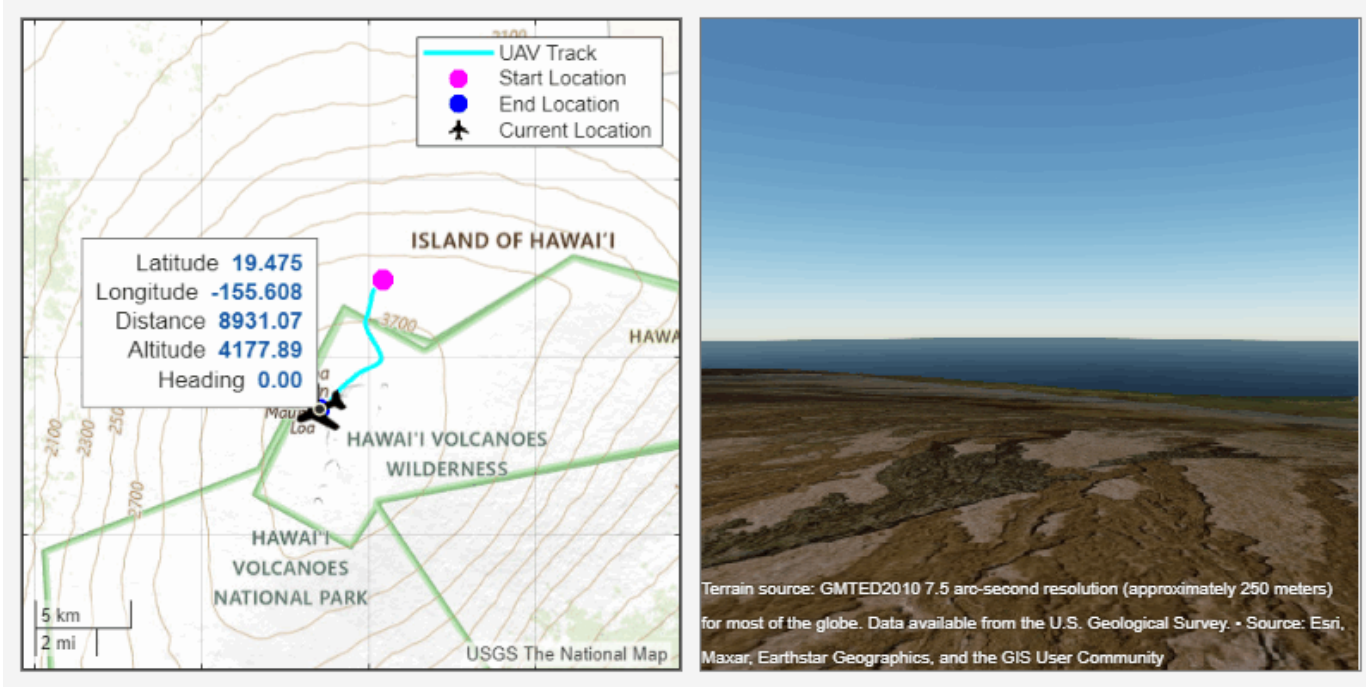
For a better view of the flight track as the UAV navigates to the top of the volcano, update the camera pitch value.

```
pitch = -2.7689;
campitch(gg,pitch)
```

Animate the flight on the 2-D map and 3-D globe.

- View the location of the UAV on the 2-D map by updating the latitude, longitude, and rotation angle of the icon. The `IconRotation` property rotates the angle in a counterclockwise-positive direction, so use the negative of the value returned by the `azimuth` function.
- View the current location, altitude, and heading of the UAV by updating the data tip to use the current index.
- Animate the 3-D flight by setting the camera position.

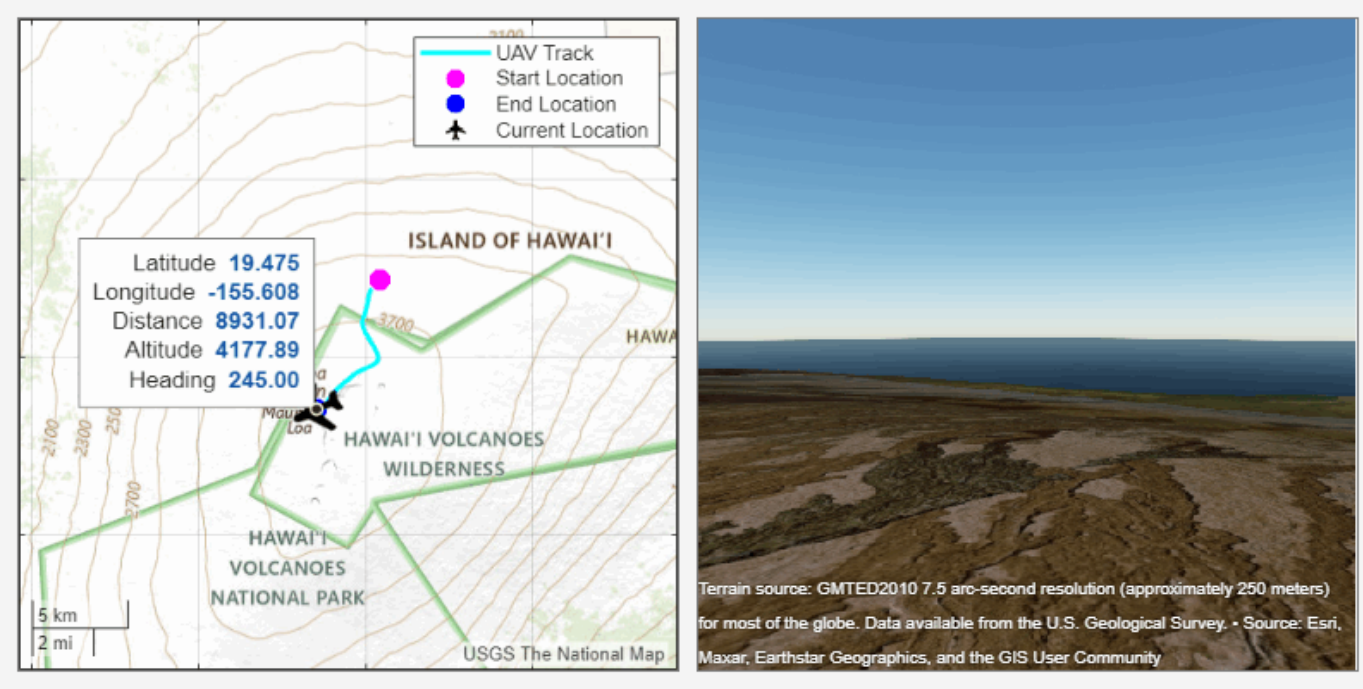
```
for k = 2:(length(tlat)-1)
    % update icon on 2-D map
    set(icon,"LatitudeData",tlat(k),"LongitudeData",tlon(k),"IconRotation",-theading(k))

    % update data tip on 2-D map
    dt.DataIndex = k;

    % update camera position for 3-D globe
    campos(gg,tlat(k),tlon(k))
    camheight(gg,talt(k)+100)
    camheading(gg,theading(k))

    drawnow
    pause(.25)
end

campos(gg,tlat(end),tlon(end),talt(end)+100)
dt.DataIndex = length(tlat);
```



View a 360-Degree Panorama from Top of Mauna Loa Volcano

View a 360-degree panorama from the top of Mauna Loa by rotating the camera heading and the icon 360 degrees. Rotate clockwise with a step size of 5-degrees and start at the next 5 degree step. Update the heading data tip.

```
initialHeading = camheading(gg);
increment = 5;
initialHeading = initialHeading + (increment - mod(initialHeading,increment));

for degree = initialHeading:increment:initialHeading+360
    heading = mod(degree,360);
    camheading(gg,heading);
    icon.IconRotation = -heading;
    ptrack.DataTipTemplate.DataTipRows(end).Value(dt.DataIndex) = heading;
    drawnow
end
```



Local Functions

Convert Height (in meters above WGS84 ellipsoid) to Zoom Level

```
function zoomLevel = heightToZoomLevel(height, lat)
    earthCircumference = 2 * pi * 6378137;
    zoomLevel = log2((earthCircumference *cosd(lat)) / height) + 1;
    zoomLevel = max(0, zoomLevel);
    zoomLevel = min(19, zoomLevel);
end
```

References

[1] Williams, Sarah C. P. “Studying Volcanic Eruptions with Aerial Drones.” *Proceedings of the National Academy of Sciences of the United States of America* 110, no. 27 (July 2, 2013): 10881. https://doi.org/10.1073/pnas.1309922110.

[2] NOAA. “Mauna Loa Baseline Observatory.” Global Monitoring Laboratory. Accessed June 16, 2020. https://gml.noaa.gov/obop/mlo/.

[3] USGS. “Mauna Loa.” Hawaiian Volcano Observatory. Accessed June 16, 2020. https://www.usgs.gov/volcanoes/mauna-loa.

See Also

Functions
`azimuth` | `campos` | `camroll` | `camheading` | `campitch` | `egm96geoid`

Objects
`GeographicGlobe`

Properties
GeographicAxes Properties

More About

- “Find Ellipsoidal Height from Orthometric Height” on page 8-27

Add Light Source to Terrain Map

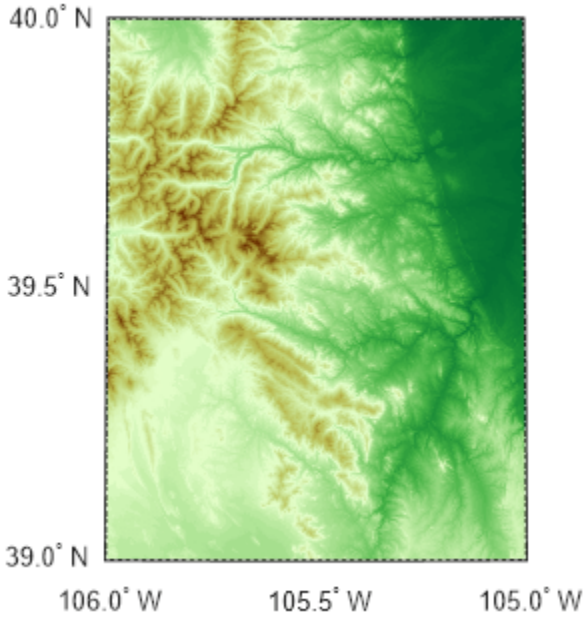
Add a light source to a terrain map for a region around South Boulder Peak in Colorado.

First, import elevation data and a geographic postings reference object. To plot the data using `geoshow`, the raster data must be of type `double` or `single`. Therefore, specify the data type for the raster using the `'OutputType'` name-value pair.

```
[Z,R] = readgeoraster('n39_w106_3arc_v2.dt1','OutputType','double');
```

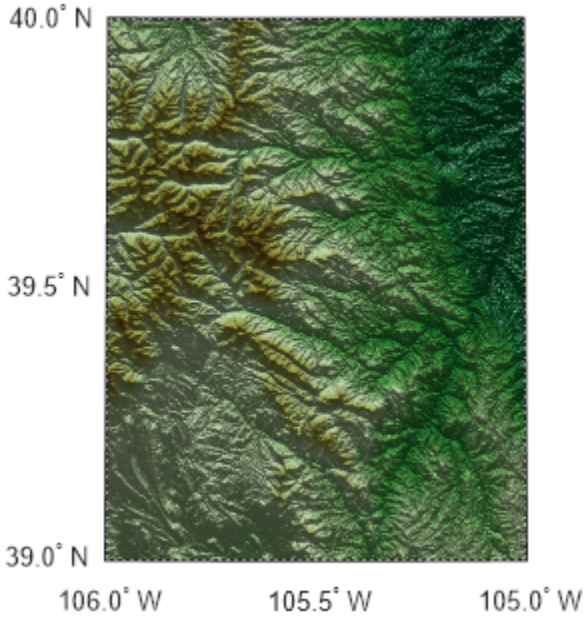
Then, display the data as a surface. Apply a colormap appropriate for terrain data using the `demcmap` function.

```
usamap(R.LatitudeLimits,R.LongitudeLimits);
geoshow(Z,R,'DisplayType','surface')
demcmap(Z)
```



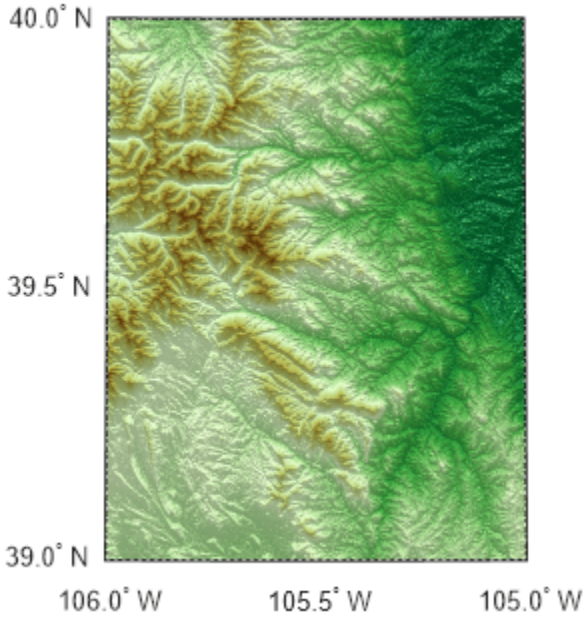
Set the vertical exaggeration using the `daspectm` function. Then, specify a light source in the top left corner of the map. Find the coordinates of the top left corner by querying the `LatitudeLimits` and `LongitudeLimits` properties of the reference object.

```
daspectm('m',20)
cornerlat = R.LatitudeLimits(2);
cornerlon = R.LongitudeLimits(1);
lightm(cornerlat,cornerlon)
```



Restore the luminance of the map by specifying the ambient, diffuse, and specular light strength.

```
ambient = 0.7;
diffuse = 1;
specular = 0.6;
material([ambient diffuse specular])
```



The DTED file used in this example is courtesy of the US Geological Survey.

See Also

`daspectm` | `lightm`

More About

- “Lighting Overview”

Surface Relief Shading

You can make dimensional monochrome shaded-relief maps with the `surflm` function, which is similar to the `surfl` function. The effect of `surflm` is similar to using lights, but the function models illumination itself (with one “light source” that you specify when you invoke it, but cannot reposition) by weighting surface normals rather than using light objects.

Shaded relief maps of this type are usually portrayed two-dimensionally rather than as perspective displays. The `surflm` function works with any projection except `globe`.

The `surflm` function accepts geolocated data grids only. Recall, however, that regular data grids are a subset of geolocated data grids, to which they can be converted using the `geographicGrid` function. The following example illustrates this procedure.

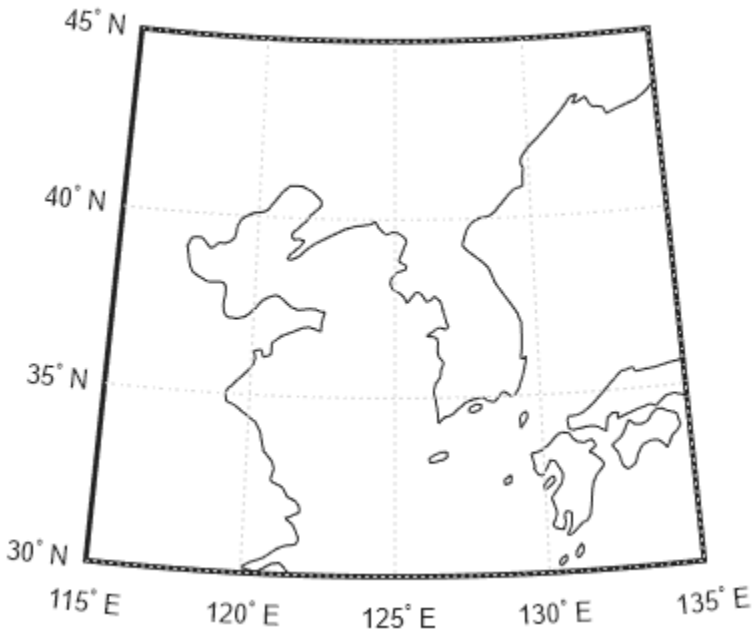
Create Monochrome Shaded Relief Map

Simulate a single light source in a figure using `surflm`. First, load elevation data and a geographic cells reference object for the Korean peninsula. Import coastline vector data using `readgeotable`. Create a map with appropriate latitude and longitude limits using `worldmap`.

```
load korea5c
latlim = korea5cR.LatitudeLimits;
lonlim = korea5cR.LongitudeLimits;
coastline = readgeotable("landareas.shp");

worldmap(latlim,lonlim)
```

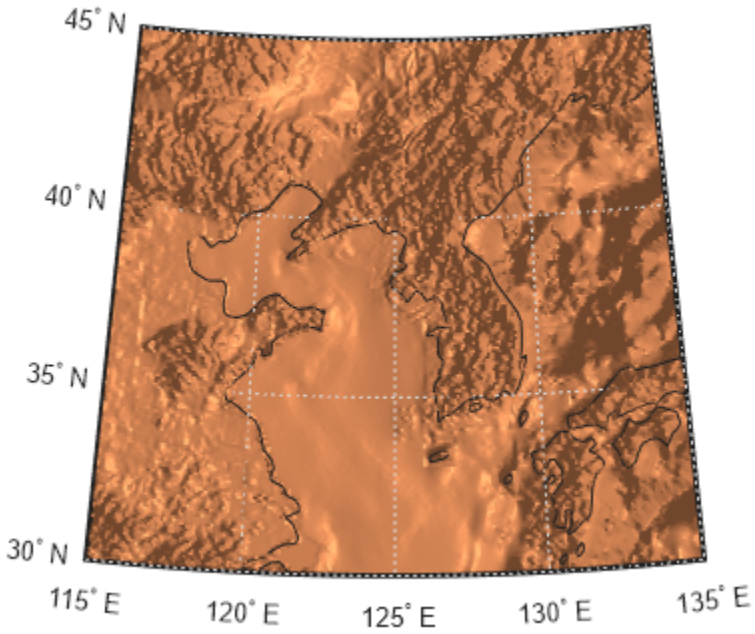
Display the coastline data using `geoshow`.

```
geoshow(coastline,'FaceColor','none')
```



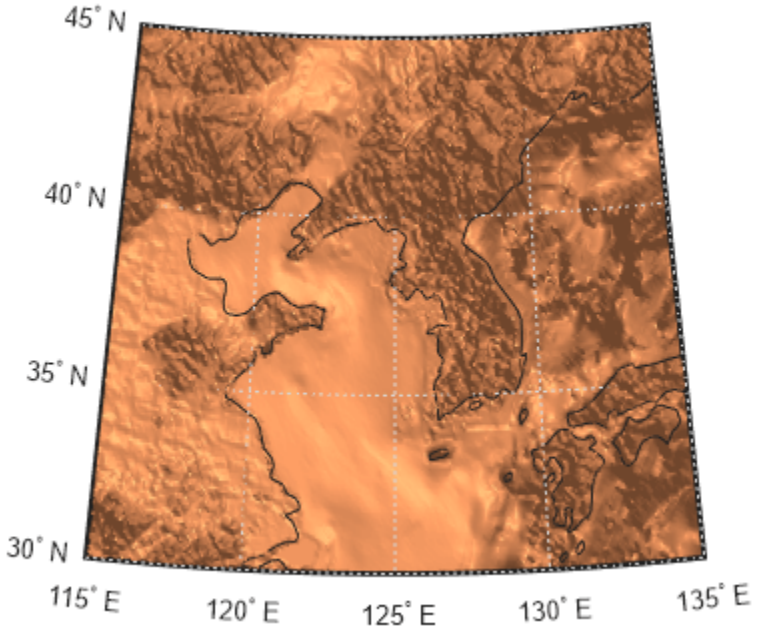
Transform the regular data grid to a geolocated data grid using the `geographicGrid` function. Then, generate a shaded relief map using `surflm`. By default, the lighting direction is 45º counterclockwise from the viewing direction. Therefore, the light source is in the southeast. Change the colormap to the monochromatic colormap `'copper'`.

```
[klat,klon] = geographicGrid(korea5cR);
s = surflm(klat,klon,korea5c);
colormap('copper')
```



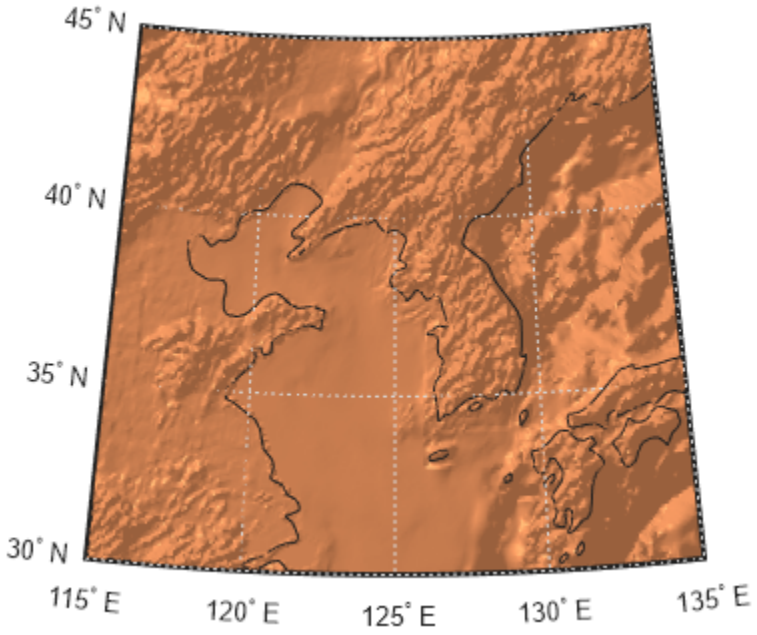
Clear the map. Then, display the shaded relief map with a different light source by specifying the azimuth as 135º and the elevation as 60º. The surface lightens and has a new character because it is lit closer to overhead and from a different direction.

```
clmo(s)
s = surflm(klat,klon,korea5c,[135 60]);
```



Shift the light to the northwest by specifying the azimuth as -135º. Lower the light to 40º above the horizon. A lower light source decreases the overall reflectance of the surface when viewed from above. Therefore, specify a 1-by-4 vector of reflectance constants that describe the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient.

```
clmo(s);
ht = surflm(klat,klon,korea5c,[-135 30],[0.65 0.4 0.3 10]);
```



The mountain ridges that run from northeast to southwest are approximately perpendicular to the light source. Therefore, these parameters demonstrate appropriate lighting for the terrain.

The monochromatic coloration in this example does not differentiate land from water. For an example that differentiates land from water, see “Colored Surface Shaded Relief” on page 5-32.

See Also

Functions
`surflm` | `geographicGrid`

Related Examples

- “Add Light Source to Terrain Map” on page 5-25
- “Relief Mapping with Light Objects” on page 5-35

Colored Surface Shaded Relief

The functions `meshlsrm` and `surflsrm` display maps as shaded relief with surface coloring as well as light source shading. You can think of them as extensions to `surflm` that combine surface coloring and surface light shading. Use `meshlsrm` to display regular data grids and `surflsrm` to render geolocated data grids.

These two functions construct a new colormap and associated `CData` matrix that uses grayscales to lighten or darken a matrix component based on its calculated surface normal to a light source. While there are no analogous MATLAB display functions that work like this, you can obtain similar results using MATLAB light objects, as “Relief Mapping with Light Objects” on page 5-35 explains.

For further information, see the reference pages for `surflsrm`, `meshlsrm`, `daspectm`, and `view`.

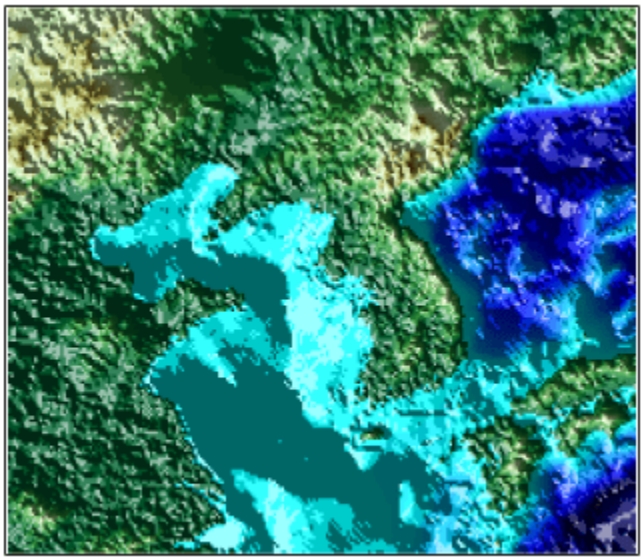
Create Colored Shaded Relief Map

Display surface illumination over colored elevation data using `surflsrm`. First, load elevation data and a geographic cells reference object for the Korean peninsula. Georeference the regular data grid using `geographicGrid`.

```
load korea5c
[klat,klon] = geographicGrid(korea5cR);
```

Create a colormap appropriate for elevation data. Plot the colored shaded relief map by specifying a light source with an azimuth of -130º and an altitude of 50º. The `surflsrm` function transforms the colormap to shade relief according to the light source. Eliminate white space around the map using `tightmap`.

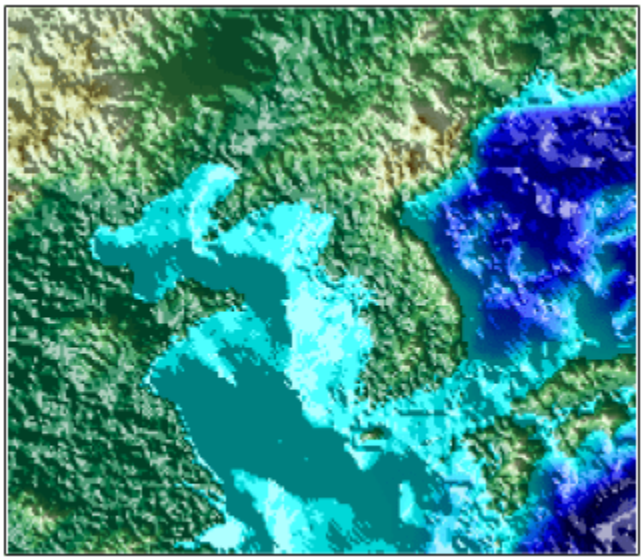
```
[cmap,clim] = demcmap(korea5c);
axesm('miller','MapLatLimit',[30 45],'MapLonLimit',[115 135])
surflsrm(klat,klon,korea5c,[-130 50],cmap,clim)
tightmap
```



You can achieve the same effect using `meshlsrm`, which operates on regular data grids.

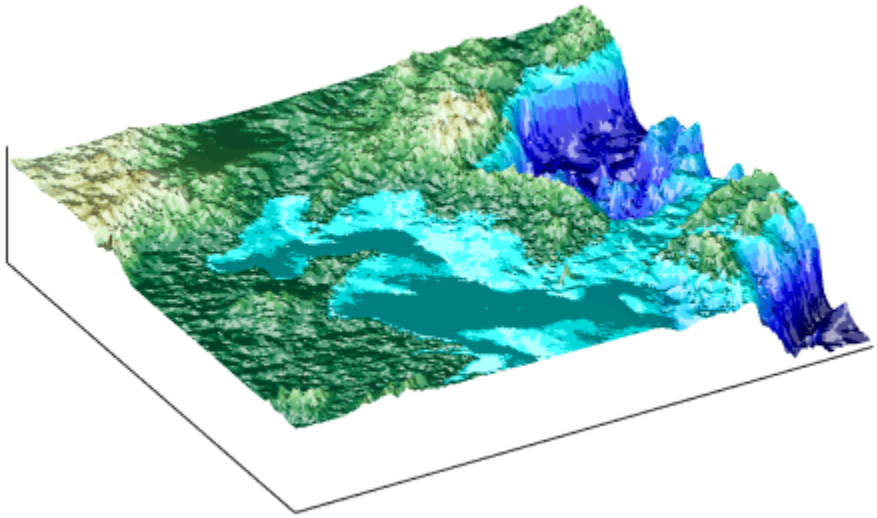
The surface has more contrast than if it were not shaded. Lighten the surface uniformly by 25%.

```
brighten(0.25)
```



Display an oblique view of the surface. Hide the bounding box by setting the `Box` property, exaggerate terrain relief by a factor of 50 using `daspectm`, and set the view to an azimuth of -30º and an altitude of 30º.

```
set(gca,'Box','off')
daspectm('meters',50)
view(-30,30)
```



You can continue rotating the perspective using `view` or the **Rotate 3D** tool in the figure window. You can continue changing the vertical exaggeration using `daspectm`. To change the built-in lighting direction, you must generate a new view using `surflsrm`.

Relief Mapping with Light Objects

This example shows how to create a light object to mimic the map produced in “Colored Surface Shaded Relief” on page 5-32, which uses shaded relief computations rather than light objects.

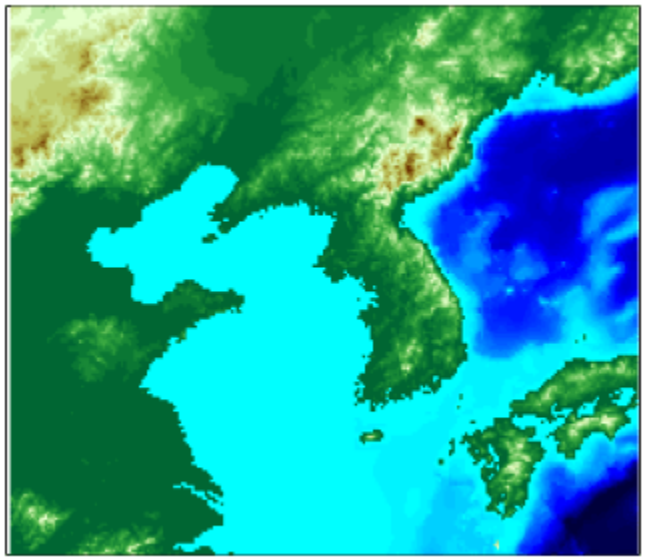
The `meshlsrm` and `surflsrm` functions simulate lighting by modifying the colormap with bands of light and dark. The map matrix is then converted to indices for the new "shaded" colormap based on calculated surface normals. Using light objects allows for a wide range of lighting effects. The toolbox manages light objects with the `lightm` function, which depends upon the MATLAB `light` function. Lights are separate MATLAB graphic objects.

For more information, consult the reference pages for `lightm`, `daspectm`, `material`, `lighting`, and `view`, along with “Lighting, Transparency, and Shading”.

Illuminate Color 3-D Relief Maps with Light Objects

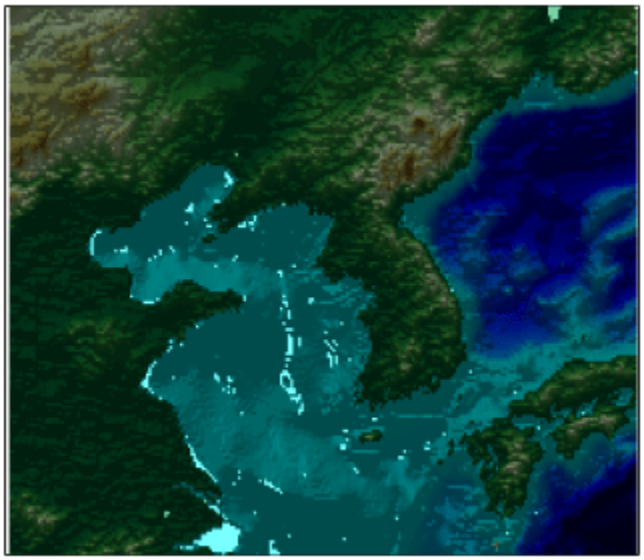
Add a light source to a surface colored data grid using `lightm`. First, load elevation data and a geographic cells reference object for the Korean peninsula. Display the data without lighting effects using `meshm`. Apply a colormap appropriate for elevation data using `demcmap`. Eliminate extra white space around the map using `tightmap`.

```
load korea5c
axesm('miller','MapLatLimit',[30 45],'MapLonLimit',[115 135])
meshm(korea5c,korea5cR,size(korea5c),korea5c)
demcmap(korea5c)
tightmap
```



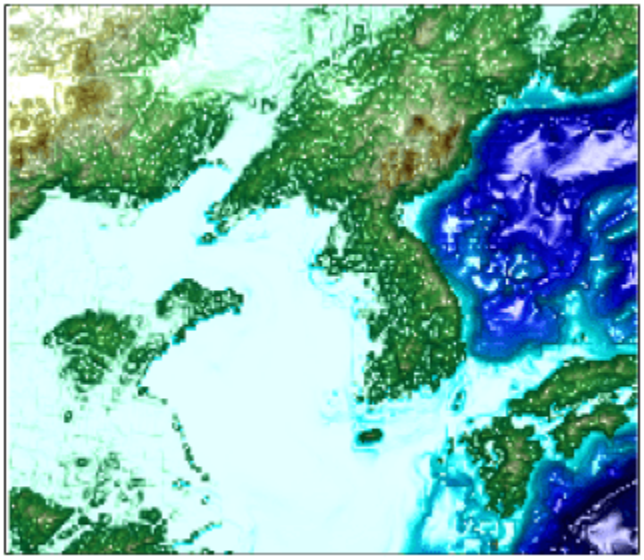
Place a light source at the northwest corner of the grid, one degree high, using `lightm`. The `lightm` function is similar to the MATLAB® function `light`, but accepts latitude and longitude inputs instead of *x*, *y*, and *z*. Note that the figure becomes darker.

```
lightm(45,115,1)
```



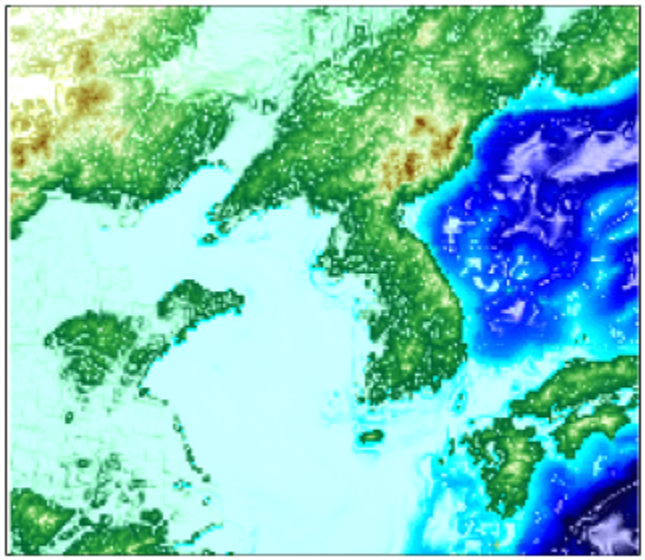
Exaggerate the vertical dimension to make any relief viewable in perspective. Note that the figure becomes darker still.

```
daspectm('meters',50)
```



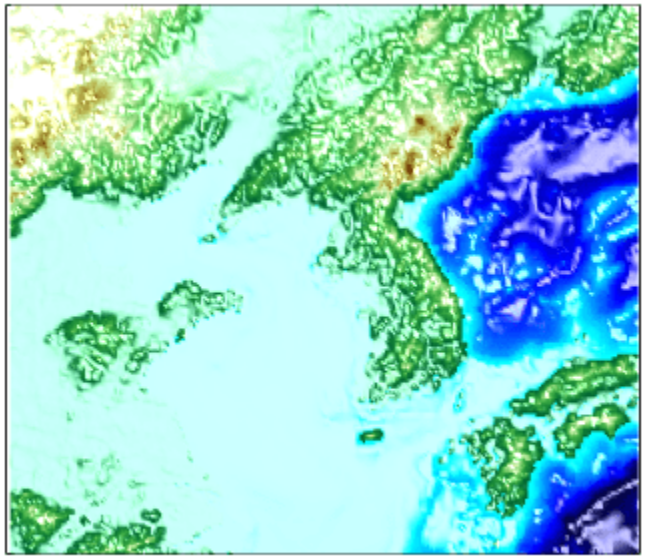
Set the ambient (direct), diffuse (skylight), and specular (highlight) surface reflectivity characteristics, respectively.

```
material([0.7 0.9 0.8])
```



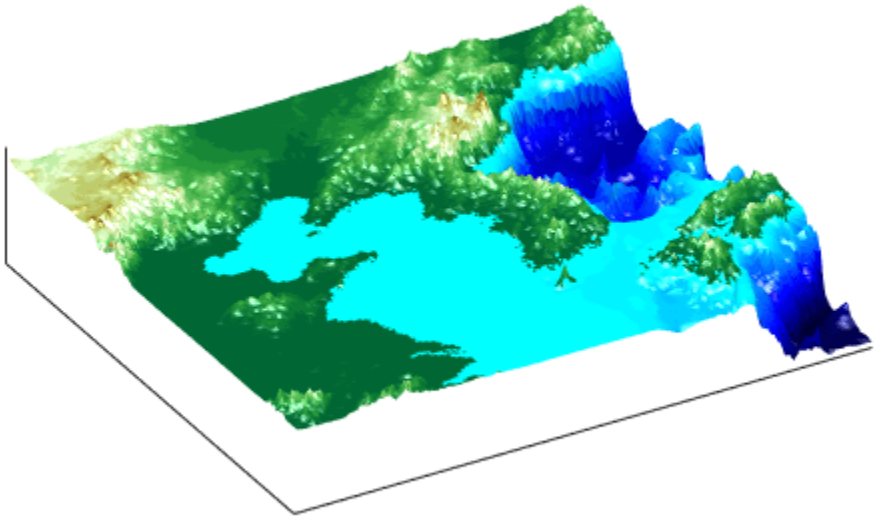
By default, the lighting is flat (plane facets). Change the light to use Gouraud shading (interpolated normal vectors at facet corners).

```
lighting Gouraud
```



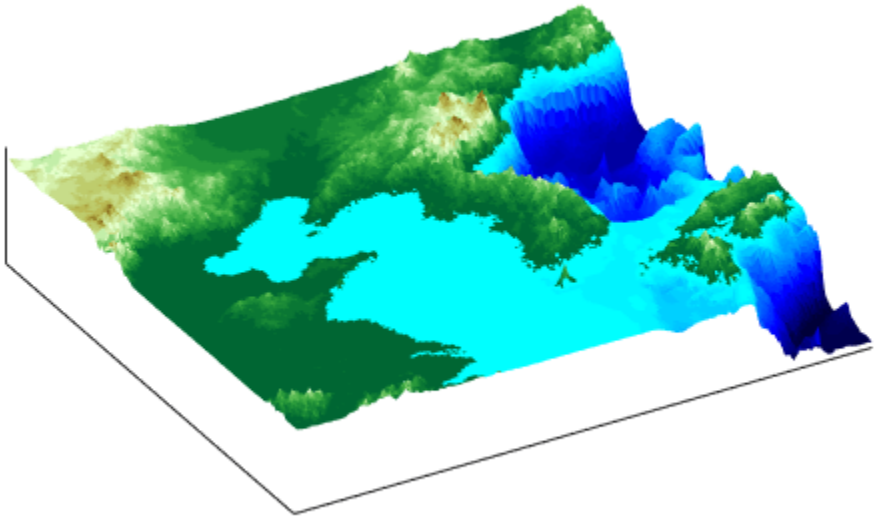
Remove the edges of the bounding box. Change the view by specifying an azimuth of -30º and an altitude of 30º.

```
ax = gca;
ax.Box = 'off';
view(-30,30)
```



If there is only one light in the current figure, you can remove it using `clmo`.

```
clmo(handlem('light'))
```



Drape Data on Elevation Maps

Combine Elevation Maps with Other Kinds of Data

Lighting effects can provide important visual cues when elevation maps are combined with other kinds of data. The shading resulting from lighting a surface makes it possible to "drape" satellite data over a grid of elevations. It is common to use this kind of display to overlay georeferenced land cover images from Earth satellites such as LANDSAT and SPOT on topography from digital elevation models.

When the elevation and image data grids correspond pixel-for-pixel to the same geographic locations, you can build up such displays using the optional altitude arguments in the surface display functions. If they do not, you can interpolate one or both source grids to a common mesh.

Drape Data over Terrain with Different Gridding

If you want to combine elevation and attribute (color) data grids that cover the same region but are gridded differently, you must resample one matrix to be consistent with the other. That is, you can construct a geolocated grid version of the regular data grid values or construct a regular grid version of the geolocated data grid values.

It helps if at least one of the grids is a geolocated data grid, because their explicit horizontal coordinates allow them to be resampled using the `geointerp` function. These examples show how to drape data over terrain with different gridding.

Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid

This example shows how to combine an elevation data grid and an attribute (color) data grid that cover the same region but are gridded differently. The example drapes slope data from a regular data grid on top of elevation data from a geolocated data grid. The example uses the geolocated data grid as the source for surface elevations and transforms the regular data grid into slope values, which are then sampled to conform to the geolocated data grid (creating a set of slope values for the diamond-shaped grid) and color-coded for surface display. This approach works with any dissimilar grids, although the two data sets in this example actually have the same origin (the geolocated grid derives from the `topo60c` data set).

Load the geolocated data grid from the `mapmtx` file and the regular data grid from the `topo60c` file. The `mapmtx` file actually contains two regions but this example only uses the diamond-shaped portion, `lt1`, `lg1`, and `map1`, centered on the Middle East.

```
load mapmtx lt1 lg1 map1
load topo60c
```

Compute the surface aspect, slope, and gradients for `topo60c`. This example only uses the slopes in subsequent steps.

```
[aspect,slope,gradN,gradE] = gradientm(topo60c,topo60cR);
```

Use the `geointerp` function to interpolate slope values to the geolocated grid specified by `lt1` and `lg1`. The output is a 50-by-50 grid of elevations matching the coverage of the `map1` variable.

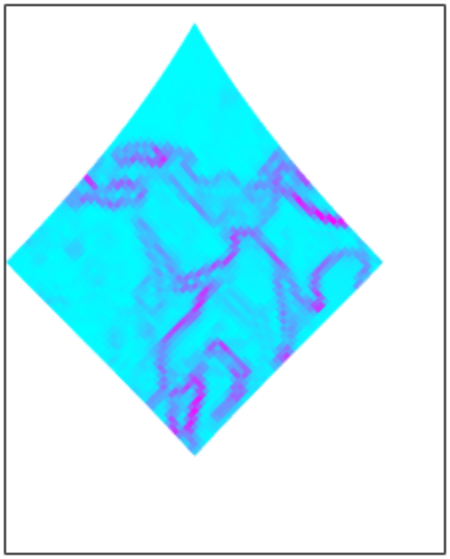
```
slope1 = geointerp(slope,topo60cR,lt1,lg1);
```

Set up a figure with a Miller projection and use `surfm` to display the slope data. Specify the *z* -values for the surface explicitly as the `map1` data, which is terrain elevation. The map mainly depicts steep cliffs, which represent mountains (the Himalayas in the northeast), and continental shelves and trenches.

```
figure
axesm miller
surfm(lt1,lg1,slope1,map1)
```

The coloration depicts steepness of slope. Change the colormap to make the steepest slopes magenta, the gentler slopes dark blue, and the flat areas light blue:

```
colormap cool
```



Use `view` to get a southeast perspective of the surface from a low viewpoint. In 3-D, you can see the topography as well as the slope.

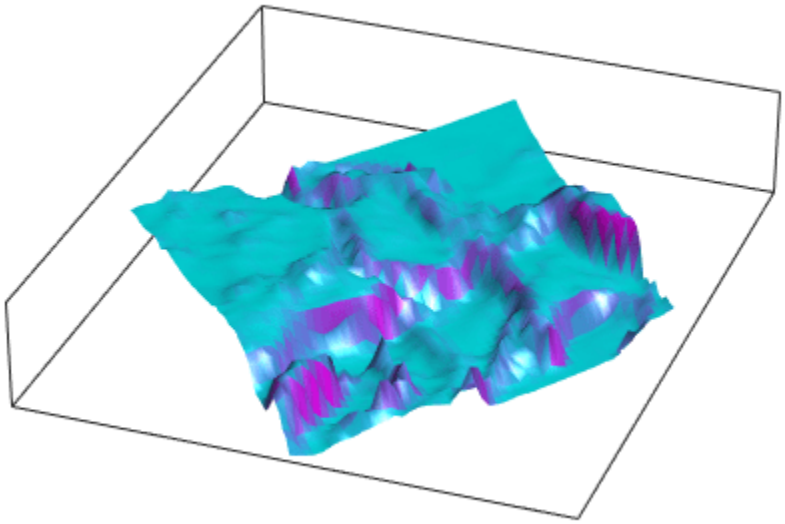
```
view(20,30)
daspectm('meter',200)
```

The default rendering uses faceted shading (no smooth interpolation). Make the surface shiny with Gouraud shading and specify lighting from the east (the default of `camlight` lights surfaces from over the right shoulder of the viewer).

```
material shiny
camlight
lighting Gouraud
```

Remove the white space and view the figure in perspective mode.

```
axis tight
ax = gca;
ax.Projection = 'perspective';
```



Drape Geolocated Grid on Regular Data Grid via Texture Mapping

This example shows how to create a new regular data grid that covers the region of the geolocated data grid, then embed the color data values into the new matrix. The new matrix might need to have somewhat lower resolution than the original, to ensure that every cell in the new map receives a value. The example combines dissimilar data grids by creating a new regular data grid that covers the region of the geolocated data grid's z-data. This approach has the advantage that more computational functions are available for regular data grids than for geolocated ones. Color and elevation grids do not have to be the same size. If the resolutions of the two data grids are different, you can create the surface as a three-dimensional elevation map and later apply the colors as a texture map. You do this by setting the surface `CData` property to contain the color matrix, and setting the surface face color to `'texturemap'`.

Load elevation raster data and a geographic cells reference object from `topo60c.mat`. Get individual variables containing terrain data from `mapmtx.mat`.

```
load topo60c
load mapmtx lt1 lg1 map1
```

Identify the geographic limits of the geolocated grid that was loaded from `mapmtx`.

```
latlim(1) = 2*floor(min(lt1(:))/2);
lonlim(1) = 2*floor(min(lg1(:))/2);
latlim(2) = 2*ceil(max(lt1(:))/2);
lonlim(2) = 2*ceil(max(lg1(:))/2);
```

Crop the global elevation data to the rectangular region enclosing the smaller grid.

```
[topo1,topo1R] = geocrop(topo60c,topo60cR,latlim,lonlim);
```

Allocate a regular grid filled uniformly with `-Inf`, to receive texture data.

```
L1R = georefcells(latlim,lonlim,2,2);
L1 = zeros(L1R.RasterSize);
L1 = L1 - Inf;
```

Overwrite `L1` using `imbedm`, converting it from a geolocated grid to a regular grid, in which the values come from the discrete Laplacian of the elevation grid `map1`.

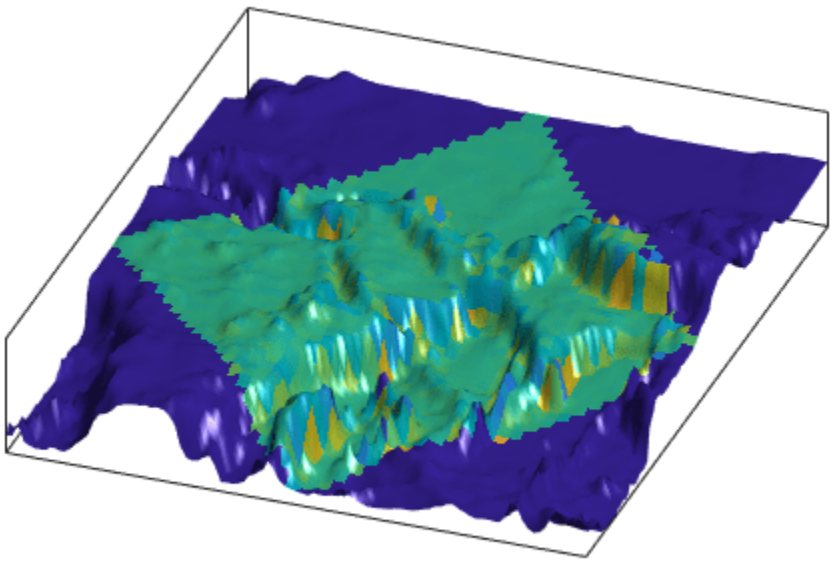
```
L1 = imbedm(lt1,lg1,del2(map1),L1,L1R);
```

Set up an `axesm`-based map with the Miller projection and use `meshm` to display the cropped data. Render the figure as a 3-D view from a 20 degree azimuth and 30 degree altitude, and exaggerate the vertical dimension by a factor of 200. Both the surface relief and coloring represent topographic elevation.

```
figure
axesm miller
h = meshm(topo1,topo1R,size(topo1),topo1);
view(20,30)
daspectm('m',200)
```

Apply the `L1` matrix as a texture map directly to the surface using the `set` function. The area not covered by the `[lt1,lg1,map1]` geolocated data grid appears dark blue because the corresponding elements of L1 were set to `-Inf`.

```
h.CData = L1;
h.FaceColor = 'texturemap';
material shiny
camlight
lighting gouraud
axis tight
```



See Also

Functions
`geointerp` | `georefcells` | `imbedm`

Related Examples

- “Drape Geoid Heights over Topography” on page 5-44

Drape Geoid Heights over Topography

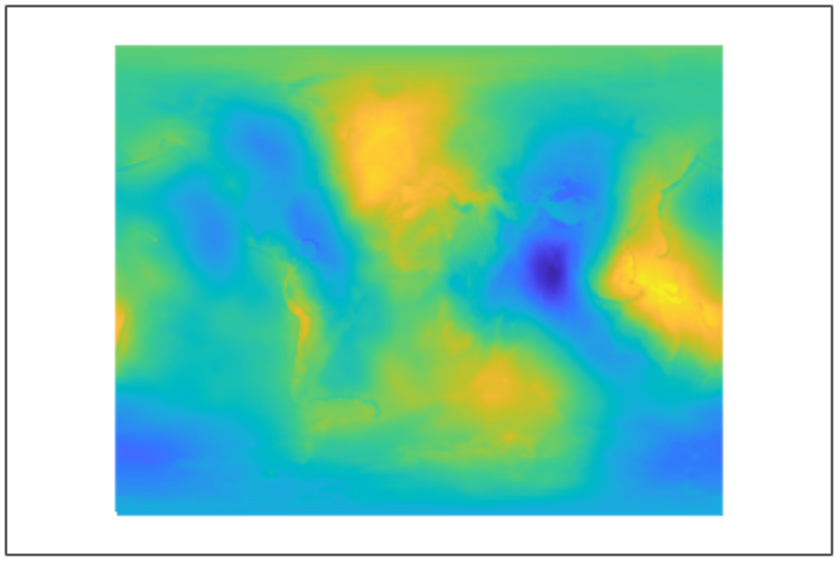
Display geoid data draped over topographic relief. For this example, display the geoid data as a color attribute instead of a 3-D surface.

Load topographic raster data and a geographic cells reference object. Get geoid heights by calling the `egm96geoid` function and specifying the reference object.

```
load topo60c
[N,R] = egm96geoid(topo60cR);
```

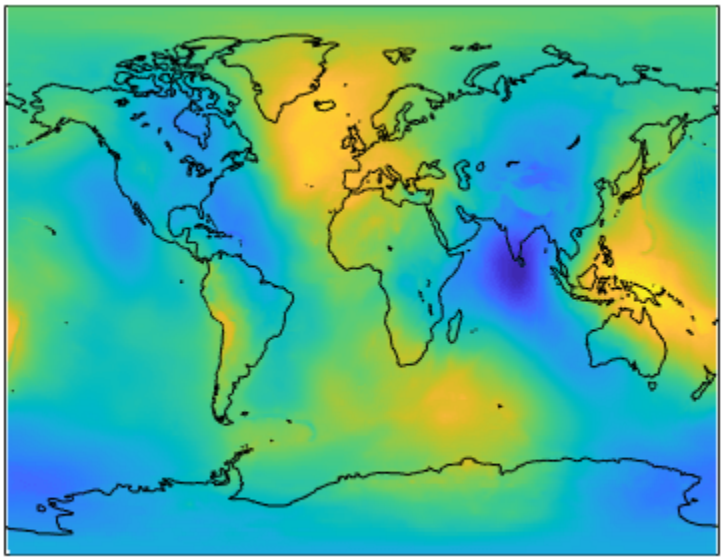
Create an `axesm`-based map using a Gall stereographic cylindrical projection (a perspective projection). Use `meshm` to display variations of the geoid in color, but specify `topo60c` as the final argument, to give each geoid grid cell the height (*z* value) of the corresponding topographic grid cell. Low geoid heights are shown as blue, high ones as yellow.

```
axesm gstereo;
meshm(N,R,size(N),topo60c)
```



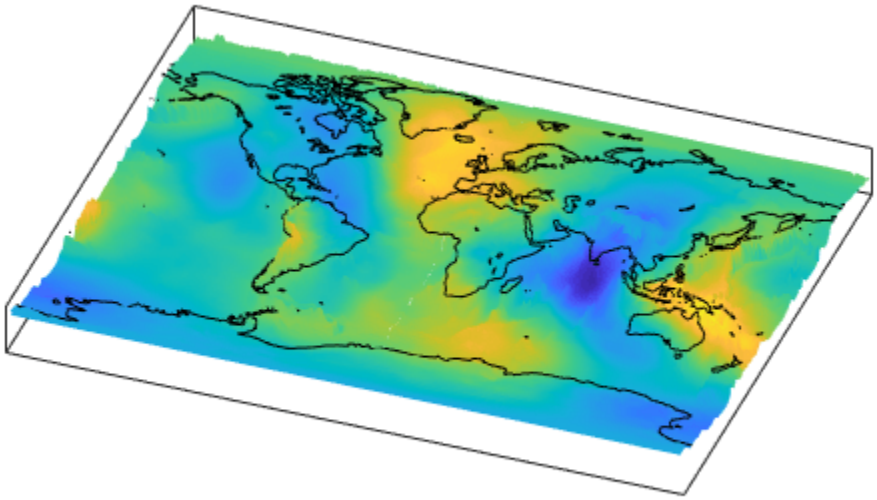
For reference, plot the world coastlines in black, raise their elevation to 1000 meters (high enough to clear the surface in their vicinity), and expand the map to fill the frame.

```
load coastlines
plotm(coastlat,coastlon,'k')
zdatam(handlem('allline'),1000)
tightmap
```



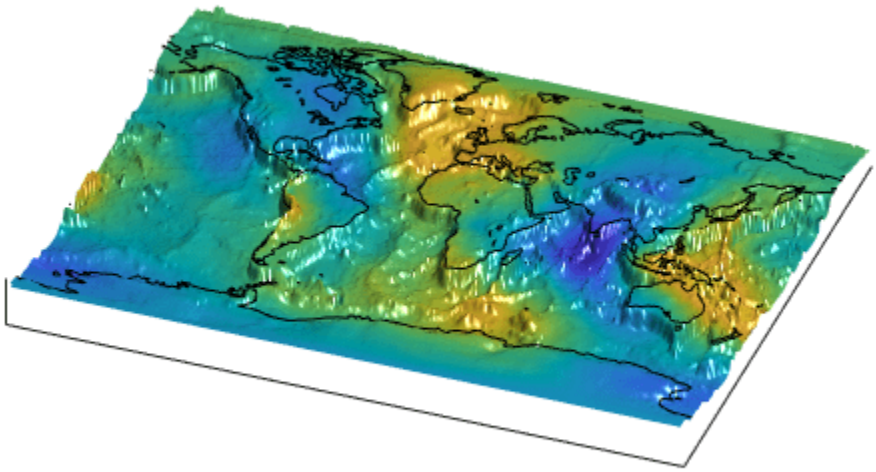
Due to the vertical view and lack of lighting, the topographic relief is not visible, but it is part of the figure's surface data. Bring it out by exaggerating relief greatly, and then setting a view from the south-southeast.

```
daspectm('m',200); tightmap
view(20,35)
```



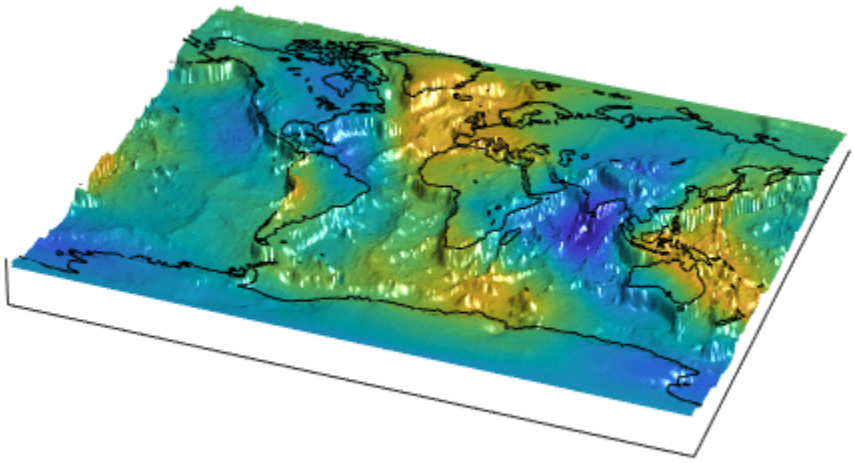
Remove the bounding box, shine a light on the surface (using the default position, offset to the right of the viewpoint), and render again with Gouraud shading.

```
ax = gca;
ax.Box = 'off';
camlight;
lighting Gouraud
```



Finally, set the perspective to converge slightly (the default perspective is orthographic). Notice that the geoid mirrors the topography of the major mountain chains such as the Andes, the Himalayas, and the Mid-Atlantic Ridge. You can also see that large areas of high or low geoid heights are not simply a result of topography.

```
ax.Projection = 'perspective';
```



Data Analysis

- “Create Geospatial Tables” on page 6-2
- “Geographic Data Structures” on page 6-7
- “Combine Vector Data Using Table Joins” on page 6-14
- “Create and Display Polygons” on page 6-17
- “Compare Polygons and Line Segments” on page 6-25
- “Simplify Vector Coordinate Data” on page 6-27
- “Geographic Interpolation of Vectors” on page 6-28
- “Find Vector Intersection Points” on page 6-30
- “Geographic Statistics for Point Locations on a Sphere” on page 6-32
- “Equal-Areas in Geographic Statistics” on page 6-36
- “Georeferenced Raster Data” on page 6-39
- “Spatially Reference Imported Rasters” on page 6-41
- “Merge Multiple Raster Tiles Using Datastore” on page 6-45
- “Construct a Global Data Grid” on page 6-51
- “Geolocated Data Grids” on page 6-53
- “Geographic Interpretations of Geolocated Grids” on page 6-56
- “Data Grids as Logical Variables” on page 6-59
- “Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid” on page 6-60
- “Georeference an Image to an Orthotile Base Layer” on page 6-63
- “Unproject a Digital Elevation Model (DEM)” on page 6-73
- “Define Areas of Interest” on page 6-82
- “Clip Vector Data to Area of Interest” on page 6-91
- “Crop Raster Data to Area of Interest” on page 6-95
- “Query Points in Area of Interest” on page 6-102
- “Convert Vector Data to Raster Format” on page 6-107
- “Combine Vector and Raster Geodata on the Same Map” on page 6-111
- “Determine and Visualize Visibility Across Terrain” on page 6-113

Create Geospatial Tables

Vector data uses points, lines, polygons, and their nonspatial attributes to describe real-world events, locations and objects. For example, you can use a point and timestamp attribute to represent a tsunami event, a line and name attribute to represent a river location, and a polygon and age attribute to represent a building.

You can represent vector data by using geospatial tables. A geospatial table is a `table` or `timetable` object that contains a `Shape` variable and attribute variables.

- The `Shape` variable contains 2-D information about point, line, and polygon shapes. The `Shape` variable can contain combinations of shape types, but all the shapes must have the same coordinate reference system (CRS). The `Shape` variable must be the first variable of the table.

  Represent shapes with coordinates in geographic CRSs by using `geopointshape`, `geolineshape`, and `geopolyshape` objects. The `GeographicCRS` property of each object contains the CRS as a `geocrs` object.

  Represent shapes with coordinates in projected CRSs by using `mappointshape`, `maplineshape`, and `mappolyshape` objects. The `ProjectedCRS` property of each object contains the CRS as a `projcrs` object.
- The attribute variables contain data such as names, classifications, and measurements.

For example, you can import the shapefile `concord_roads.shp`, which represents a road network in Concord, MA, as a geospatial table. Each row of the table contains a `maplineshape` object with information about the line shape and several attributes with information such as the street name, road class, and road length.

There are multiple ways to create geospatial tables. You can:

- Read a geospatial table from a vector data file by using the `readgeotable` function.
- Read a table or timetable from a file containing pointwise data or well-known text (WKT) string representations of geometry by using the `readtable` or `readtimetable` function, and then convert the table to a geospatial table by using the `table2geotable` function.
- Create a geospatial table by using the `table` or `timetable` function. For more information about creating tables, see “Create Tables and Assign Data to Them”. For more information about creating timetables, see “Create Timetables”.

These examples show how to create geospatial tables using each of the listed processes.

Read Geospatial Table from File

Read a shapefile containing a network of road segments in Concord, MA as a geospatial table by using the `readgeotable` function.

```
GT = readgeotable("concord_roads.shp");
```

View the first eight rows of the geospatial table. The `Shape` variable contains information about the shapes. For this example, all the shapes are lines. The other variables contain attribute data.

```
head(GT)
```

Shape	STREETNAME	RT_NUMBER	CLASS	ADMIN_TYPE	LENGTH
maplineshape	""	""	6	0	67.264
maplineshape	"WRIGHT FARM"	""	5	0	72.178
maplineshape	"WRIGHT FARM"	""	5	0	43.965
maplineshape	"WRIGHT FARM"	""	5	0	109.65
maplineshape	"WRIGHT FARM"	""	5	0	18.019
maplineshape	""	""	6	0	58.444
maplineshape	"WRIGHT FARM"	""	5	0	16.925
maplineshape	""	""	6	0	55.633

View the projected CRS for the line shapes. All shapes in a geospatial table must have the same CRS.

```
GT.Shape.ProjectedCRS
```

```
ans =
  projcrs with properties:

                   Name: "NAD83 / Massachusetts Mainland"
          GeographicCRS: [1×1 geocrs]
       ProjectionMethod: "Lambert Conic Conformal (2SP)"
             LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

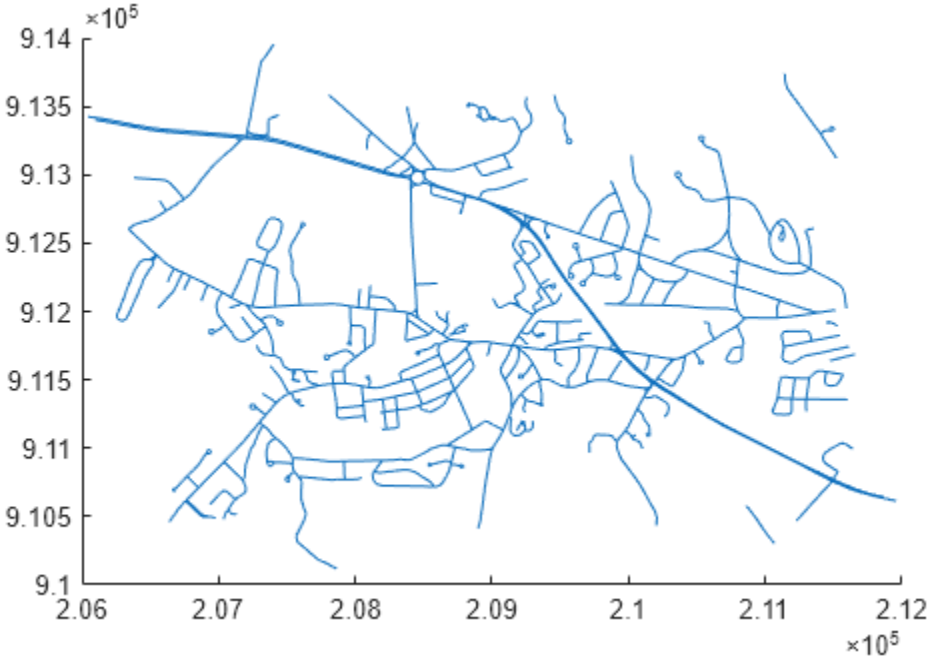
View information about the line shape in the first row of the table.

```
GT.Shape(1)
```

```
ans =
  maplineshape with properties:

                NumParts: 1
                Geometry: "line"
    CoordinateSystemType: "planar"
            ProjectedCRS: [1×1 projcrs]
```

Display the roads.

```
mapshow(GT)
```



Create Geospatial Table from Table

Read data about tsunami events as a table by using the `readtable` function. The coordinates of the tsunami source locations are in the `Latitude` and `Longitude` table variables.

```
T = readtable("tsunamis.xlsx");
```

Convert the table to a geospatial table by using the `table2geotable` function. The function detects the `Latitude` and `Longitude` variables and uses them to create the `Shape` variable of the table.

```
GT = table2geotable(T);
```

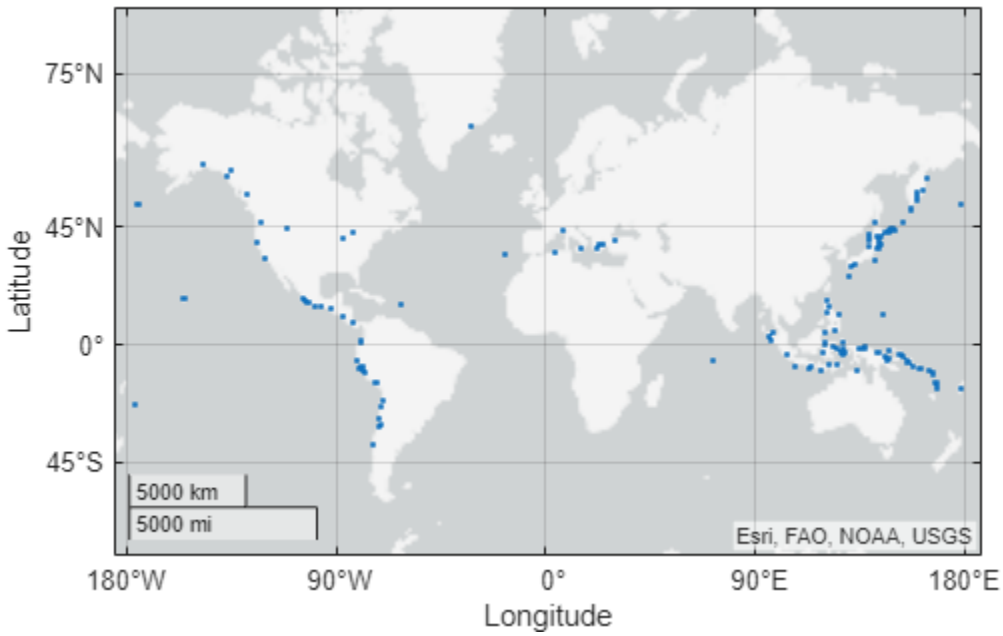
View the `Shape` variable of the geospatial table. The tsunami source locations are points in a geographic coordinate system.

```
GT.Shape
```

```
ans =
  162×1 geopointshape array with properties:

              NumPoints: [162×1 double]
               Latitude: [162×1 double]
              Longitude: [162×1 double]
               Geometry: "point"
    CoordinateSystemType: "geographic"
           GeographicCRS: []
```

Plot the source locations on a map.

```
geoplot(GT)
```



Create Geospatial Table from Arrays

Create a geospatial table containing the locations and names of cities.

Create an array of `geopointshape` objects from column vectors of latitude and longitude coordinates. Specify the geographic CRS as the World Geodetic System of 1984, which has the EPSG code `4326`.

```
lats = [35.7082 -22.8842 51.5074 39.9042 37.9838]';
lons = [139.6401 -43.3882 -0.1278 116.4074 23.7275]';
shape = geopointshape(lats,lons);
shape.GeographicCRS = geocrs(4326)
```

```
shape =
  5×1 geopointshape array with properties:

               NumPoints: [5×1 double]
                Latitude: [5×1 double]
               Longitude: [5×1 double]
                Geometry: "point"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Specify the names of the cities as a column vector.

```
names = ["Tokyo" "Rio de Janeiro" "London" "Beijing" "Athens"]';
```

Create the geospatial table. The `Shape` variable contains the `geopointshape` objects and the `Name` variable contains the names.

```
GT = table(shape,names,VariableNames=["Shape" "Name"])
```

```
GT=5×2 table
             Shape                    Name
    ________________________    _______________

    (35.7082°N, 139.6401°E)    "Tokyo"
    (22.8842°S,  43.3882°W)    "Rio de Janeiro"
    (51.5074°N,   0.1278°W)    "London"
    (39.9042°N, 116.4074°E)    "Beijing"
    (37.9838°N,  23.7275°E)    "Athens"
```

Verify that the table is a geospatial table.

```
isgeotable(GT)
```

```
ans = logical
   1
```

Tips

Some functions that accept geospatial tables as input require the geospatial table to contain shapes of a specific type. For example, the `kmlwrite` function only accepts geospatial tables that contain geographic point, line, or polygon shapes.

See Also

Functions
`readgeotable` | `readtable` | `table2geotable` | `struct2geotable` | `isgeotable`

Objects
`table` | `timetable`

Geographic Data Structures

Vector data uses points, lines, polygons, and their nonspatial attributes to describe events, locations, and objects. One way to represent vector data is by using geographic data structures. A geographic data structure is a structure array that contains one element per geographic feature.

There are two types of geographic data structures. A mapstruct array stores map coordinates (*x* and *y*) using `X` and `Y` fields, and a geostruct array stores geographic coordinates (latitude and longitude) using `Lon` and `Lat` fields.

Note Starting in R2021b, you can also represent vector data by using geospatial tables. Geospatial tables are `table` or `timetable` objects with a `Shape` variable and attribute variables. Geospatial tables enable you to:

- Filter, rearrange, and access data using standard MATLAB table functions.
- Examine and modify the shapes by using functions such as `isinterior`, `mapclip`, and `geoclip`.
- Plot data on geographic axes or map axes by specifying tables as input to the `geoplot` function.

For more information about geospatial tables, see “Create Geospatial Tables” on page 6-2.

There are several ways to create geographic data structures:

- Read a geographic data structure from a shapefile by using the `shaperead` function.
- Read a geographic data structure from GSHHG data by using the `gshhs` function.
- Create a geographic data structure from variables in the workspace.

Fields of Geographic Data Structures

Geographic data structures contain these fields:

Field	Description
`Geometry`	Type of shape, specified as a character vector such as `'Point'`, `'Multipoint'`, `'Line'`, or `'Polygon'`. Individual points, lines, and polygons can have multiple parts. For example, lines can contain disconnected line segments, and polygons can have rings. All parts of an individual shape share the attributes specified for that shape by the corresponding attribute fields. Geographic data structures do not support assigning distinct attributes to the different parts of an individual shape.

Field	Description
`BoundingBox`	Minimum and maximum coordinate values for the shape, specified as a 2-by-2 numeric matrix of the form `[min(X) min(Y); max(X) max(Y)]`. When `Geometry` is `'Point'`, this field does not apply.
`X`	*x*-coordinates, specified as a row vector of data type `double`. This field applies to only mapstruct arrays.
`Y`	*y*-coordinates, specified as a row vector of data type `double`. This field applies to only mapstruct arrays.
`Lon`	Longitude coordinates, specified as a row vector of data type `double`. This field applies to only geostruct arrays.
`Lat`	Latitude coordinates, specified as a row vector of data type `double`. This field applies to only geostruct arrays.
Attribute fields	Attribute fields, specified as character vectors or numeric scalars.

Read Geographic Data Structures from Shapefiles

This example shows how to read data from a shapefile into a geographic data structure by using the `shaperead` function. The `shaperead` function is the most common way to create geographic data structures.

Starting in R2021b, you can read the same data into a geospatial table by using the `readgeotable` function. The `readgeotable` function supports several vector data formats in addition to shapefiles, including KML and GeoJSON files.

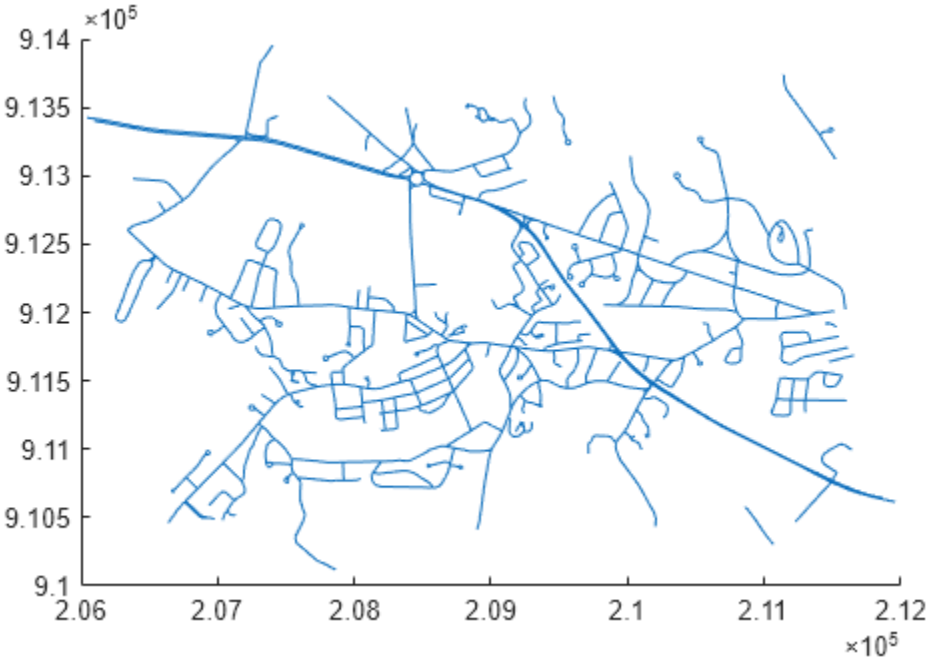
Read a shapefile containing road data into the workspace. The function returns a geographic data structure (in this case a mapstruct array) that contains 609 elements. In addition to the `Geometry`, `BoundingBox`, and coordinate fields (`X` and `Y`), the mapstruct array has five attribute fields: `STREETNAME`, `RT_NUMBER`, `CLASS`, `ADMIN_TYPE`, and `LENGTH`.

```
S = shaperead("concord_roads.shp")
```

```
S=609×1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Display the road data on a map.

```
mapshow(S)
```



By default, the `shaperead` function assumes that the shapefile contains map coordinates and reads the data into a mapstruct array. When your shapefile contains geographic coordinates, you can read the data into a geostruct array by specifying the `UseGeoCoords` name-value argument as `true`. If you do not know whether a shapefile contains map coordinates or geographic coordinates, you can try these strategies.

- If the shapefile includes a projection file (`.prj`), you can get information about the coordinate reference system by using the `shapeinfo` function. If the `CoordinateReferenceSystem` field of the returned structure is a `projcrs` object, then the shapefile contains map coordinates. If the field is a `geocrs` object, then the shapefile contains geographic coordinates.
- If the shapefile does not include a projection file, you can get the bounding box of the coordinates by using the `shapeinfo` function. You might be able to determine the type of coordinates by looking at the ranges of the coordinates.

If you cannot determine whether your shapefile contains map coordinates or geographic coordinates using these strategies, consider asking your data provider.

Create Geographic Data Structures

Create geographic data structures from variables in the workspace. The variables can represent data imported from files or data computed using MATLAB® functions.

Starting in R2021b, you can also create geospatial tables from variables in the workspace. For more information about creating geospatial tables, see “Create Geospatial Tables” on page 6-2.

Create Geostruct Array of Points

Create a geostruct array of points. For this example, use the geostruct array to store the locations and names of cities.

Specify the latitude and longitude coordinates of Paris, Santiago, and New York City.

```
latP = 48.87084;
lonP = 2.41306;
latS = -33.36907;
lonS = -70.82851;
latN =  40.69746;
lonN = -73.93008;
```

Create an empty structure array. Add a `Geometry` field that defines the structure array elements as points. Add `Lat` and `Lon` fields that store the coordinates of the points.

```
Cities = struct;

[Cities(1:3).Geometry] = deal('Point');

Cities(1).Lat = latP;
Cities(1).Lon = lonP;

Cities(2).Lat = latS;
Cities(2).Lon = lonS;

Cities(3).Lat = latN;
Cities(3).Lon = lonN;
```

Add a `Name` attribute field that stores the names of the cities.

```
Cities(1).Name = "Paris";
Cities(2).Name = "Santiago";
Cities(3).Name = "New York";
```

View the geostruct array.

```
Cities
```

```
Cities=1×3 struct array with fields:
    Geometry
    Lat
    Lon
    Name
```

Create Geostruct Array of Lines

Create a geostruct array of lines. For this example, use the geostruct array to store great circle tracks between the cities.

Create an empty structure array. Add a `Geometry` field that defines the structure array elements as lines.

```
Tracks = struct;

[Tracks(1:3).Geometry] = deal('Line');
```

Add a `Type` attribute field that identifies the tracks as great circle tracks. Many functions use the input `"gc"` to identify tracks as great circle tracks.

```
trackType = "gc";
[Tracks.Type] = deal(trackType);
```

Calculate the coordinates of the great circle tracks between the cities. Then, add the coordinates to the structure array as `Lat` and `Lon` fields.

```
[latPS,lonPS] = track2(trackType,latP,lonP,latS,lonS);
[latSN,lonSN] = track2(trackType,latS,lonS,latN,lonN);
[latNP,lonNP] = track2(trackType,latN,lonN,latP,lonP);

Tracks(1).Lat = latPS;
Tracks(1).Lon = lonPS;

Tracks(2).Lat = latSN;
Tracks(2).Lon = lonSN;

Tracks(3).Lat = latNP;
Tracks(3).Lon = lonNP;
```

Label the track names by adding a `Name` attribute field to the structure array.

```
Tracks(1).Name = "Paris-Santiago";
Tracks(2).Name = "Santiago-New York";
Tracks(3).Name = "New York-Paris";
```

Calculate the lengths and azimuths of the great circle tracks. Then, add the results to the structure array as `Length` and `Azimuth` attribute fields.

```
[distPS,azPS] = distance(trackType,latP,lonP,latS,lonS);
[distSN,azSN] = distance(trackType,latS,lonS,latN,lonN);
[distNP,azNP] = distance(trackType,latN,lonN,latP,lonP);

Tracks(1).Length = distPS;
Tracks(1).Azimuth = azPS;

Tracks(2).Length = distSN;
Tracks(2).Azimuth = azSN;

Tracks(3).Length = distNP;
Tracks(3).Azimuth = azNP;
```

View the geostruct array.

```
Tracks
```

```
Tracks=1×3 struct array with fields:
    Geometry
    Type
    Lat
    Lon
    Name
    Length
    Azimuth
```

Display Data Contained in Geostruct Arrays

Display the city locations and names on the map.

```
figure
axesm("mercator","MapLatLimit",[-75 75])
tightmap
geoshow("landareas.shp","FaceColor",[0.82 0.91 0.72],"EdgeColor","none")

geoshow(Cities,"Marker","o","MarkerFaceColor","k","MarkerEdgeColor","k")
textm([Cities.Lat],[Cities(:).Lon]+5,[Cities(:).Name])
```

Display the great circle tracks on the map. Specify the appearance of the lines by using a symbol specification.

```
colorRange = makesymbolspec("Line", ...
    {'Length',[min([Tracks.Length]) max([Tracks.Length])],"Color",lines(3)});
geoshow(Tracks,"SymbolSpec",colorRange)
```

Create Geostruct Array of Polygons

The process for creating a geostruct or mapstruct array of polygons is similar to the process for creating geostruct arrays of points and lines. For information about specifying the coordinates of polygon vertices, see “Create and Display Polygons” on page 6-17.

For an example of working with geostruct arrays of polygons, see “Convert Coastline Data (GSHHG) to Shapefile Format” on page 2-82.

See Also

Functions
`shaperead` | `shapeinfo`

More About

- “Create and Display Polygons” on page 6-17
- “Select Shapefile Data to Read” on page 2-39

Combine Vector Data Using Table Joins

To link vector data from one file to vector data with shared attributes from other files, import the files as tables and use a table join. A *table join* combines two tables by row according to key variables such as state names, alphanumeric IDs, or Federal Information Processing System (FIPS) codes.

This example shows you how to read data from two different files into separate tables, join the tables, create a choropleth map from the joined table, and export the joined table into a shapefile.

Read Data into Tables

Read a comma-separated text file into a table. The file contains farmland statistics for each US state and the District of Columbia, based on the National Agricultural Statistics Service (NASS) Farms and Land in Farms 2019 Summary. For more information about the statistics, see [1] on page 6-16. Display the first row of the table.

```
farms = readtable("farmland.txt",TextType="string");
farms(1,:)
```

```
ans=1×4 table
    StateName    NumberOfFarms    AcresInFarms    AverageFarmSize
    _________    _____________    ____________    _______________

    "Alabama"        38800            8300              214
```

To create a map from the table, you also need spatial information. Read a shapefile containing polygon shapes for each of the US states and the District of Columbia into a geospatial table. A *geospatial table* has a `Shape` variable containing shape information for each row. Display the first row of the geospatial table.

```
states = readgeotable("usastatelo.shp");
states(1,:)
```

```
ans=1×5 table
       Shape          Name       LabelLat    LabelLon    PopDens2000
    ____________    _________    ________    ________    ___________

    geopolyshape    "Alabama"     32.283     -86.921        87.6
```

The `Shape` variable of the table contains `geopolyshape` objects, which are polygons in geographic coordinates. Display the properties of the `geopolyshape` object in the first row.

```
states.Shape(1)
```

```
ans =
  geopolyshape with properties:

              NumRegions: 1
                NumHoles: 0
                Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Join Tables

Join the tables, using the state name variables `Name` and `StateName` as the key variables.

```
statesFarms = outerjoin(states,farms,LeftKey="Name",RightKey="StateName");
```

The `outerjoin` function creates a table that includes rows where the key variables from one input table have no matches in the other input table. As a result, the output table can have more rows than either of the input tables. To create a table that includes the rows from one input table and appends rows where the key variables match values in the key variables of the other input table, use the `join` function instead. To create a table that includes only the rows where the key variables from one input table have matching values in the other input table, use the `innerjoin` function instead.

Inspect Joined Table

Mismatches in key variables can cause unexpected results in the joined table. Inspect the joined table by displaying the number of rows.

```
height(statesFarms)
```

```
ans =
52
```

Each input table contains 51 rows, but the joined table contains 52 rows. Because the state names are the key variables, this result indicates a mismatch in the names.

The table of farmland statistics identifies the District of Columbia as Washington DC, which does not match the corresponding row of the table of polygon shapes. Change the name in the table of farmland statistics so that it matches the name in the table of polygon shapes.

```
dcRow = farms.StateName == "Washington DC";
farms.StateName(dcRow) = "District of Columbia";
```

Join the tables again and display the number of rows in the joined table.

```
statesFarms = outerjoin(states,farms,LeftKey="Name",RightKey="StateName");
height(statesFarms)
```

```
ans =
51
```

The number of rows in the output table matches the number of rows in the input table.

Sort the joined table by average farm size and store the top eight rows in a subtable.

```
statesFarms8 = topkrows(statesFarms,8,"AverageFarmSize");
```

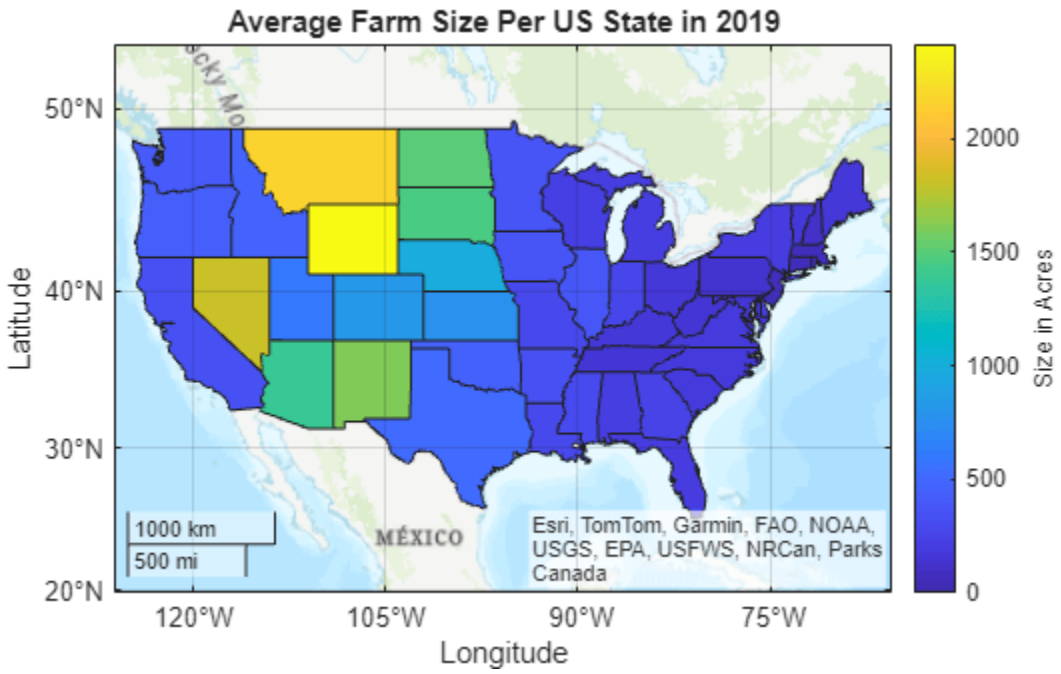
Create Map

Create a choropleth map of average farm size for the conterminous US.

```
figure
rows = statesFarms.Name ~= "Hawaii" & statesFarms.Name ~= "Alaska";
geoplot(statesFarms(rows,:),ColorVariable="AverageFarmSize")
geobasemap topographic
```

Add a title and colorbar.

```
title("Average Farm Size Per US State in 2019")
cb = colorbar;
cb.Label.String = "Size in Acres";
```



Export Table as Shapefile

Export the joined table as a new shapefile.

```
shapewrite(statesFarms,"usastatelo_farmland.shp")
```

References

[1] National Agricultural Statistics Service. “Number of Farms, Land in Farms, and Average Farm Size — States and United States: 2018–2019.” *In Farms and Land in Farms 2019 Summary*, 6. USDA, February 2020. https://www.nass.usda.gov/Publications/Todays_Reports/reports/fnlo0220.pdf.

See Also

Functions
`readtable` | `readgeotable`

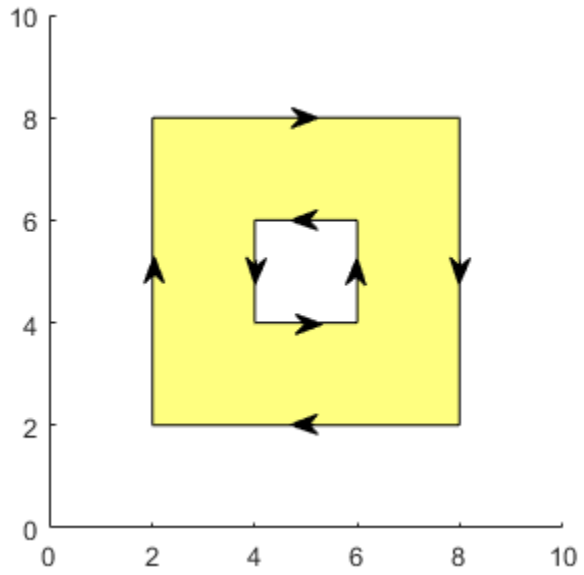
Objects
`geopolyshape`

Related Examples

- “Create Geospatial Tables” on page 6-2

Create and Display Polygons

Polygons represent geographic objects that cover area, such as continents, islands, and lakes. They may contain holes or multiple regions. Create a polygon by listing vertices that define its boundaries without intersecting. The order of the vertices determines what parts of the polygon are filled. List external boundaries clockwise and internal boundaries counterclockwise, such that the inside of the polygon is always to the right of the boundary.



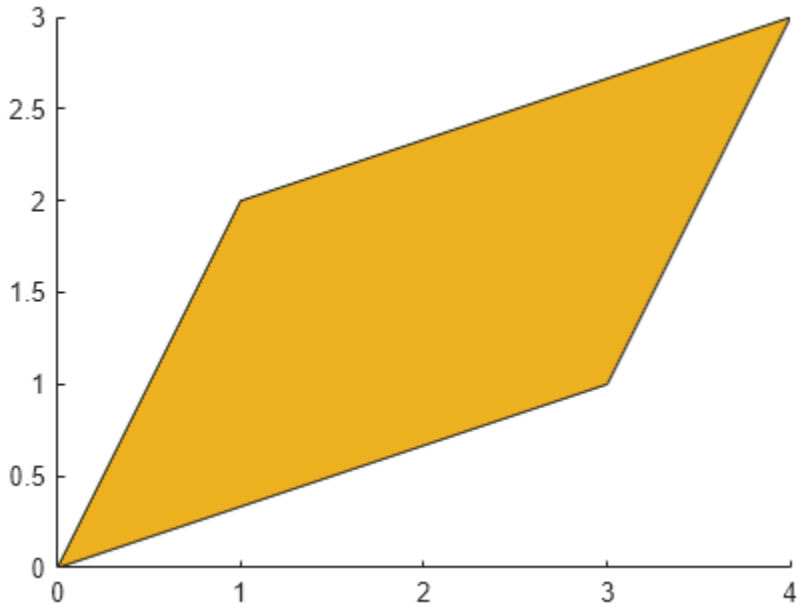
Simple Polygon

Display a simple polygon with one region and no holes. First, list its vertices in a clockwise order. Close the polygon by repeating the first vertex at the end of the list.

```
x1 = [0 3 4 1 0];
y1 = [0 1 3 2 0];
```

Display the vertices as a polygon using the `mapshow` function by specifying `'DisplayType'` as `'polygon'`.

```
mapshow(x1,y1,'DisplayType','polygon')
```



Polygons with Holes or Multiple Regions

Define polygons with multiple regions or holes by separating the boundaries with `NaN` values. List the vertices of external boundaries in a clockwise order and the vertices of internal boundaries in a counterclockwise order.

```
x2 = [0 1 8 6 0 NaN 1 4 2 1 NaN 5 6 7 3 5];
y2 = [0 6 8 2 0 NaN 1 3 5 1 NaN 3 5 7 6 3];
```

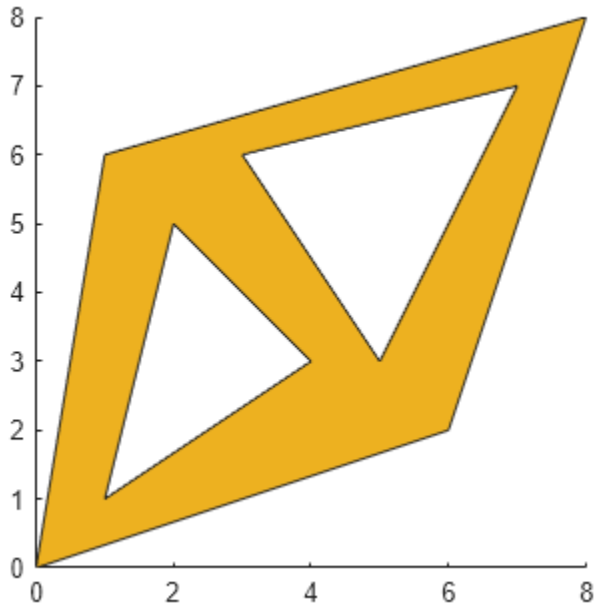
These vectors define a polygon with one external boundary and two internal boundaries. The boundaries are separated using `NaN` values. Verify the vertex order of the boundaries using the `ispolycw` function. The `ispolycw` function returns 1 when the vertices are in a clockwise order.

```
ispolycw(x2,y2)
```

```
ans = 1×3 logical array

   1   0   0
```

Display the polygon. The internal boundaries create holes within the polygon.

```
figure
mapshow(x2,y2,'DisplayType','polygon')
```



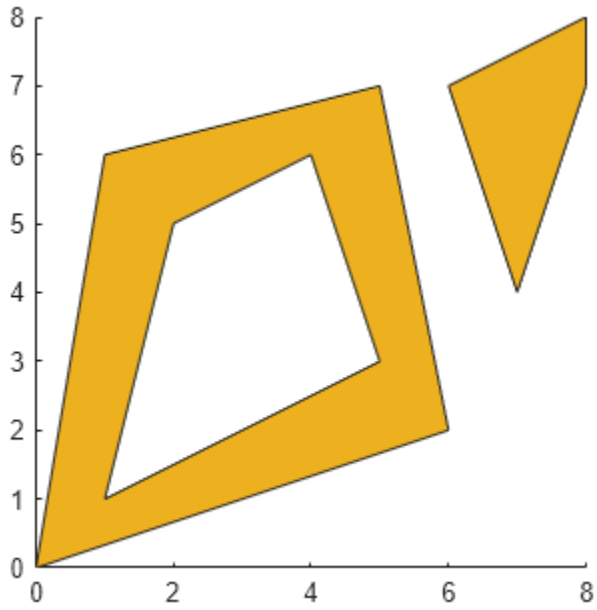
Now, list the vertices for a polygon with two nonintersecting regions. One of the regions has a hole. Verify the vertex order of the boundaries using `ispolycw`.

```
x3 = [0 1 5 6 0 NaN 1 5 4 2 1 NaN 7 6 8 8 7];
y3 = [0 6 7 2 0 NaN 1 3 6 5 1 NaN 4 7 8 7 4];
ispolycw(x3,y3)
```

```
ans = 1×3 logical array

   1   0   1
```

Display the polygon. The external boundaries create two nonintersecting regions and the internal boundary creates a hole.

```
figure
mapshow(x3,y3,'DisplayType','polygon')
```



Polygons Using Geographic Coordinates

In general, you can use geographic coordinates when you define polygons over small regions and call functions such as `ispolycw`. This is true except in cases where the polygon wraps a pole or crosses the Antimeridian.

For example, display the state of Michigan on a map using polygons with geographic coordinates. First, read the vertices of the state boundaries.

```
states = shaperead('usastatehi.shp','UseGeoCoords',true);
michigan = states(22);
lat = michigan.Lat;
lon = michigan.Lon;
```

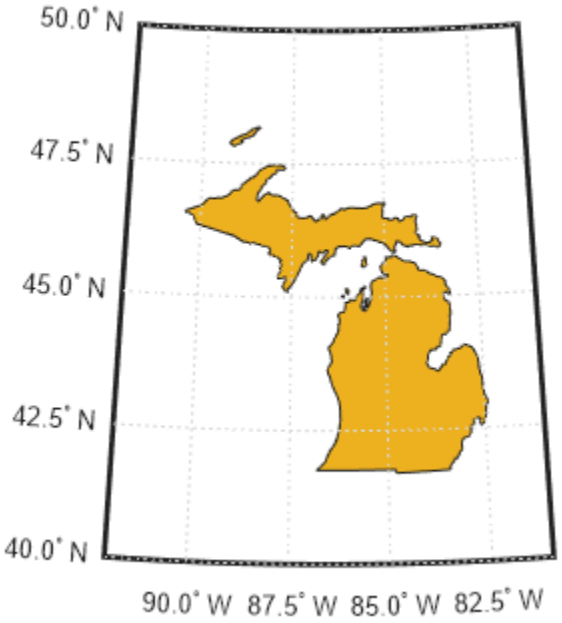
Count the boundaries and verify their vertex order. To use `ispolycw` with geographic coordinates, list the longitude vector as the first argument and the latitude vector as the second argument. The 1-by-6 output array means there are six boundaries. Each element of the array is 1, which means that each boundary is the exterior boundary of its own region.

```
ispolycw(lon,lat)
```

```
ans = 1×6 logical array

   1   1   1   1   1   1
```

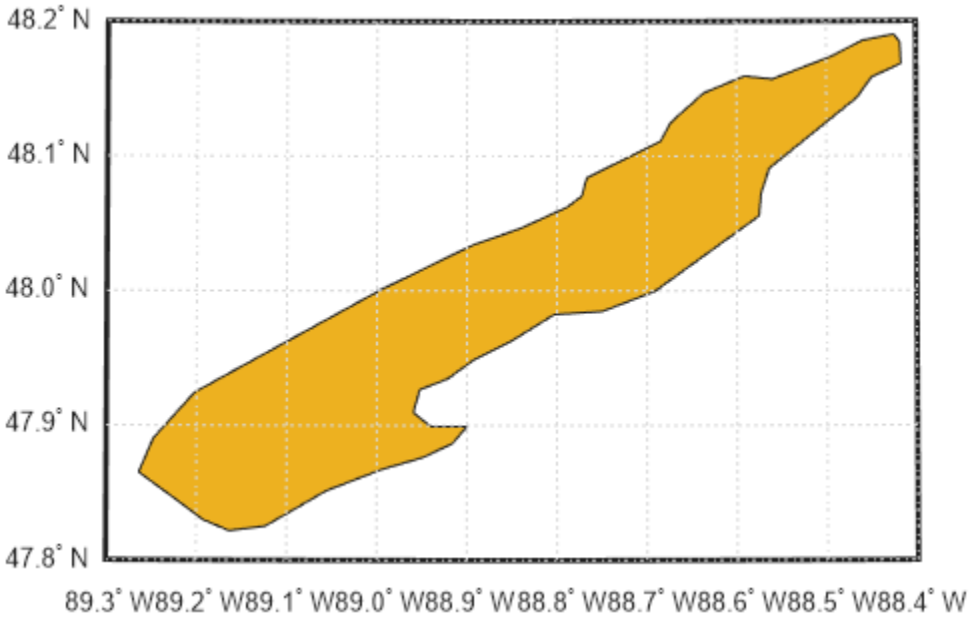
Display the polygon on a map using the `geoshow` function, specifying `'DisplayType'` as `'polygon'`.

```
usamap 'Michigan'
geoshow(lat,lon,'DisplayType','polygon')
```



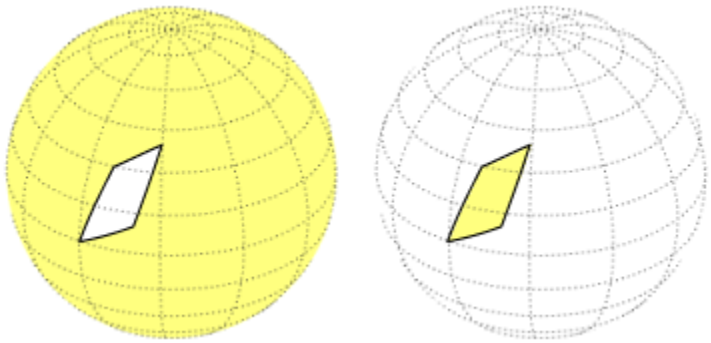
Clip the polygon to the latitude and longitude limits of Isle Royale National Park using the `maptrimp` function. Display the clipped polygon on a new map.

```
latlim = [47.8 48.2];
lonlim = [-89.3 -88.4];
[latT,lonT] = maptrimp(lat,lon,latlim,lonlim);

figure
usamap(latlim,lonlim)
geoshow(latT,lonT,'DisplayType','polygon')
```



Filled Region of Polygons Using Geographic Coordinates

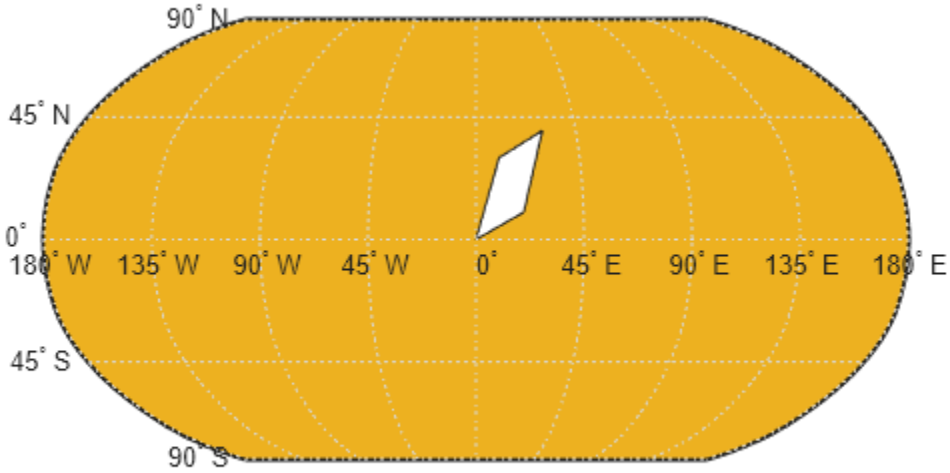
When you display a polygon on the Earth, the boundary divides the Earth into two regions. Both of these regions have finite area, so either could be the inside region of the polygon.



As a result, when you project the vertices of a polygon onto a map using the `geoshow` function, the filled region may be different than you expect. Change which region is filled by reversing the order of the vertices.

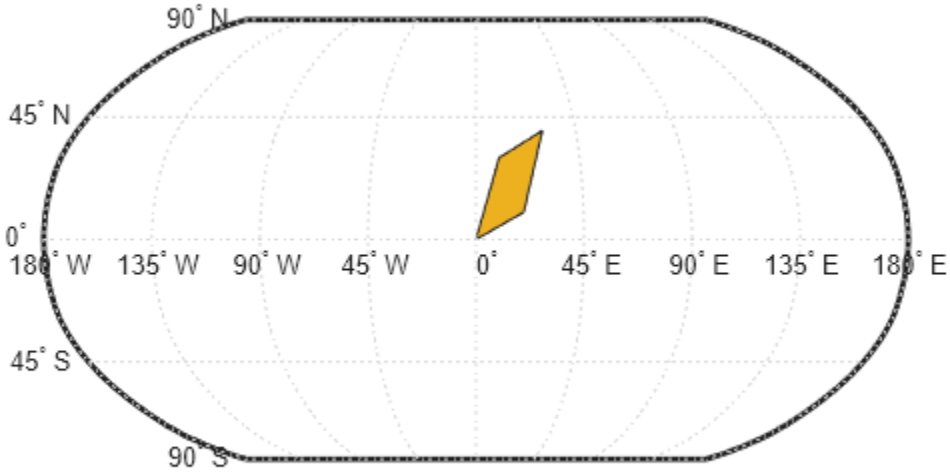
For example, display a small polygon on a world map.

```
lat2 = [0 10 40 30 0];
lon2 = [0 20 30 10 0];

figure
worldmap('world')
geoshow(lat2,lon2,'DisplayType','polygon')
```



The outside region of the polygon is filled. Reverse the order of the vertices by applying the `flip` function to the coordinate vectors. Then, display the polygon again.

```
lat2f = flip(lat2);
lon2f = flip(lon2);

figure
worldmap('world')
geoshow(lat2f,lon2f,'DisplayType','polygon')
```



The inside region of the polygon is filled instead.

See Also

Functions
`ispolycw` | `worldmap` | `geoshow` | `mapshow`

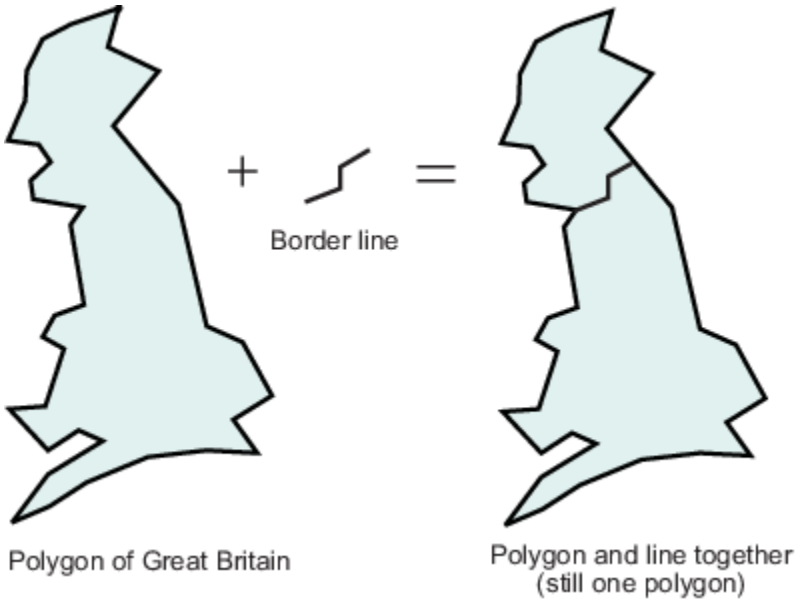
Objects
`polyshape` | `mappolyshape` | `geopolyshape`

Compare Polygons and Line Segments

Geographic objects represented by vector data might or might not be formatted as polygons. Imagine two variables, `latcoast` and `loncoast`, that correspond to a sequence of points that caricature the coast of the island of Great Britain. If this data returns to its starting point, then a polygon describing Great Britain exists. This data might be plotted as a patch or as a line, and it might be logically employed in calculations as either.

Now suppose that you want to represent the Anglo-Scottish border, proceeding from the west coast at Solway Firth to the east coast at Berwick-upon-Tweed. This data can only be properly defined as a line, defined by two or more points, which you can represent with two more variables, `latborder` and `lonborder`. When plotted together, the two pairs of variables can form a map. The patch of Great Britain plus the line showing the Scottish border might look like two patches or regions, but there is no object that represents England and no object that represents Scotland, either in the workspace or on the map.

In order to represent both regions properly, the Great Britain polygon needs to be split at the two points where the border meets it, and a copy of `latborder` and `lonborder` concatenated to both lines (placing one in reverse order). The resulting two polygons can be represented separately (e.g., in four variables named `latengland`, `lonengland`, `latscotland`, and `lonscotland`) or in two variables that define two polygons each, delineated by NaNs (e.g., `latuk`, `lonuk`).



The distinction between line and polygon data might not appear to be important, but it can make a difference when you are performing geographic analysis and thematic mapping. For example, polygon data can be treated as line data and displayed with functions such as `linem`, but line data cannot be handled as polygons unless it is restructured to make all objects close on themselves.

See Also

`geoshow` | `polymerge`

More About

- “Create and Display Polygons” on page 6-17

Simplify Vector Coordinate Data

Avoiding visual clutter in composing maps is an essential part of cartographic presentation. In cartography, this is described as map generalization, which involves coordinating many techniques, both manual and automated. Limiting the number of points in vector geodata is an important part of generalizing maps, and is especially useful for conditioning cartographic data, plotting maps at small scales, and creating versions of geodata for use at small scales.

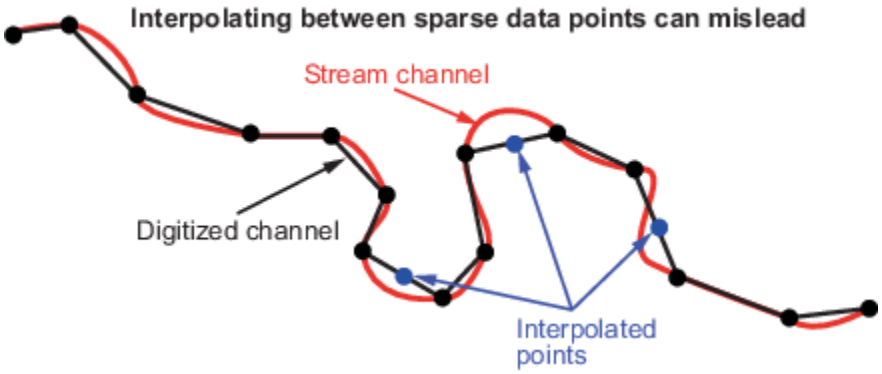
An easy, but naive, approach to point reduction is to discard every *n*th element in each coordinate vector (simple decimation). However, this can result in poor representations of the original shapes. The toolbox provides a function to eliminate insignificant geometric detail in linear and polygonal objects, while still maintaining accurate representations of their shapes. The `reducem` function implements a powerful line simplification algorithm (known as Douglas-Peucker) that intelligently selects and deletes visually redundant points.

The `reducem` function takes latitude and longitude vectors, plus an optional linear tolerance parameter as arguments, and outputs reduced (simplified) versions of the vectors, in which deviations perpendicular to local "trend lines" in the vectors are all greater than the tolerance criterion. Endpoints of vectors are preserved. Optional outputs are an error measure and the tolerance value used (it is computed when you do not supply a value).

Note Simplified line data might not always be appropriate for display. If all or most intermediate points in a feature are deleted, then lines that appear straight in one projection can be incorrectly displayed as straight lines in others, and separate lines can be caused to intersect. In addition, when you are reducing data over large world regions, the effective degree of reduction near the poles are less than that achieved near the equator, due to the fact that the algorithm treats geographic coordinates as if they were planar.

Geographic Interpolation of Vectors

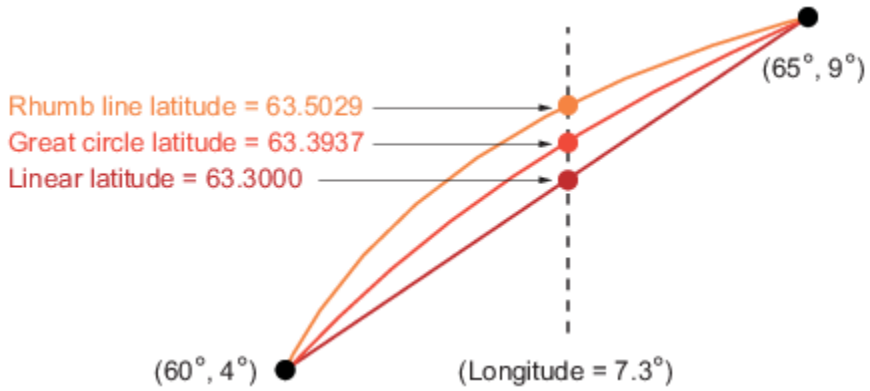
When using vector data, remember that, like raster data, coordinates are sampled measurements. This involves unavoidable assumptions concerning what the geographic reality is between specified data points. The normal assumption when plotting vector data requires that points be connected with straight line segments, which essentially indicates a lack of knowledge about conditions between the measured points. For lines that are by nature continuous, such as most rivers and coastlines, such piecewise linear interpolation can be false and misleading, as the following figure depicts.



Interpolating Sparse Vector Data

Despite the possibility of misinterpretation, circumstances do exist in which geographic data interpolation is useful or even necessary. To do this, use the `interpm` function to interpolate between known data points. One value of linearly interpolating points is to fill in lines of constant latitude or longitude (e.g., administrative boundaries) that can curve when projected.

`interpm` returns both the original data and new linearly interpolated points. Sometimes, however, you might want only the interpolated values. The functions `intrplat` and `intrplon` work similarly to the MATLAB `interp1` function, and give you control over the method used for interpolation. Note that they only interpolate and return one value at a time. Use `intrplat` to interpolate a latitude for a given longitude. Given a monotonic set of longitudes and their matching latitude points, you can interpolate a new latitude for a longitude you specify, interpolating along linear, spline, cubic, rhumb line, or great circle paths. The longitudes must increase or decrease monotonically. If this is not the case, you might be able to use the `intrplon` companion function if the latitude values are monotonic. The following diagram illustrates these three types of interpolation. The `intrplat` function also can perform spline and cubic spline interpolations.



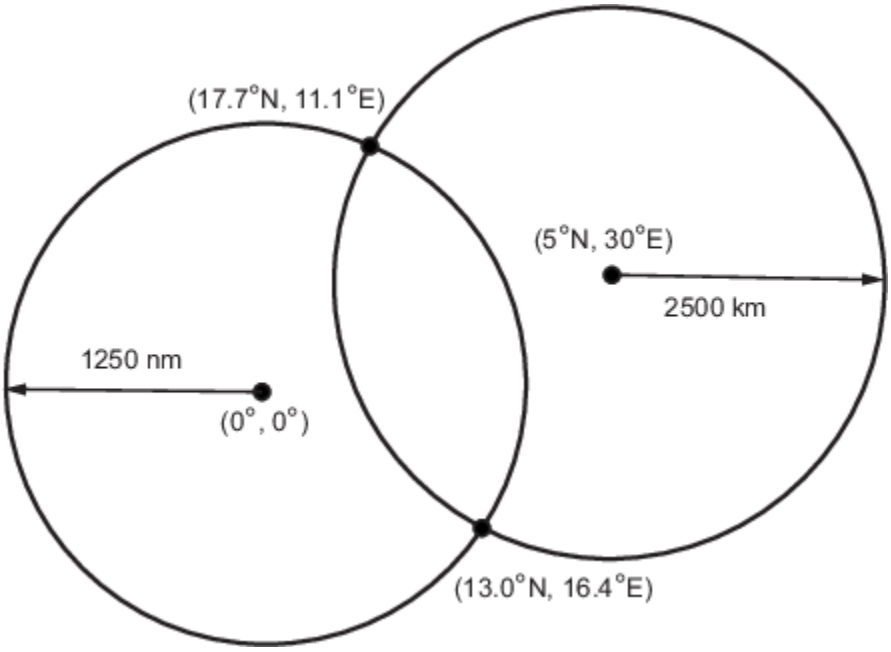
Three Types of Interpolation

Find Vector Intersection Points

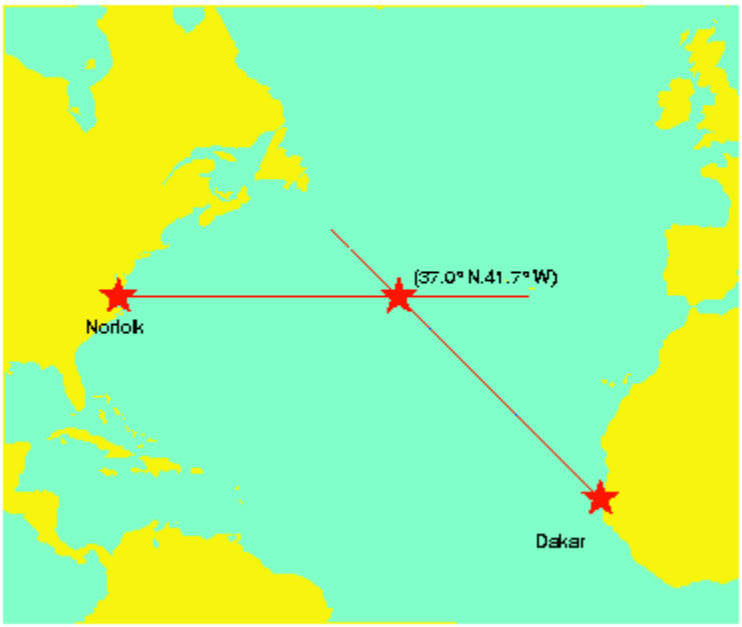
Mapping Toolbox includes a set of functions that calculate the intersections of lines, polygons, and circles that are represented by coordinate arrays. The functions also determine intersections of arbitrary vector data.

Function	Description
`gcxgc`	Find intersection points for pairs of great circles on the sphere
`scxsc`	Find intersection points for pairs of small circles on the sphere
`rhxrh`	Find intersection points, if any, for pairs of rhumb lines
`gcxsc`	Find intersection points, if any, between a great circle and a small circle on the sphere
`polyxpoly`	Find intersection points for lines or polygon edges in the plane

In general, small circles intersect twice or never, as shown in the following figure. For the case of exact tangency, `scxsc` returns two identical intersection points.



To illustrate finding the intersection of rhumb lines, imagine a ship setting sail from Norfolk, Virginia (37°N,76°W), maintaining a steady due-east course (90°), and another ship setting sail from Dakar, Senegal (15°N,17°W), with a steady northwest course (315°). Where would the tracks of the two vessels cross? The intersection of the tracks is at (37°N,41.7°W), which is roughly 600 nautical miles west of the Azores in the Atlantic Ocean.


(37.0° N,41.7° W)
Norfolk
Dakar

Geographic Statistics for Point Locations on a Sphere

Certain Mapping Toolbox functions compute basic geographical measures for spatial analysis and for filtering and conditioning data. Since MATLAB functions can compute statistics such as means, medians, and variances, why not use those functions in the toolbox? First of all, classical statistical formulas typically assume that data is one-dimensional (and, often, normally distributed). Because this is not true for geospatial data, spatial analysts have developed statistical measures that extend conventional statistics to higher dimensions.

Second, such formulas generally assume that data occupies a two-dimensional Cartesian coordinate system. Computing statistics for geospatial data with geographic coordinates as if it were in a Cartesian framework can give statistically inappropriate results. While this assumption can sometimes yield reasonable numerical approximations within small geographic regions, for larger areas it can lead to incorrect conclusions because of distance measures and area assumptions that are inappropriate for spheres and spheroids. Mapping Toolbox functions appropriately compute statistics for geospatial data, avoiding these potential pitfalls.

Geographic Means

Consider the problem of calculating the mean position of a collection of geographic points. Taking the arithmetical mean of the latitudes and longitudes using the standard MATLAB `mean` function may seem reasonable, but doing this could yield misleading results.

Take two points at the same latitude, 180° apart in longitude, for example (30°N,90°W) and (30°N,90°E). The *mean* latitude is (30+30)/2=30, which seems right. Similarly, the mean longitude must be (90+(-90))/2=0. However, as one can also express 90°W as 270°E, (90+270)/2=180 is also a valid mean longitude. Thus there are two correct answers, the prime meridian and the dateline. This demonstrates how the sphericity of the Earth introduces subtleties into spatial statistics.

This problem is further complicated when some points are at different latitudes. Because a degree of longitude at the Arctic Circle covers a much smaller distance than a degree at the equator, distance between points having a given difference in longitude varies by latitude.

Is in fact 30°N the right mean latitude in the first example? The mean position of two points should be equidistant from those two points, and should also minimize the total distance. Does (30°N,0°) satisfy these criteria?

```
dist1 = distance(30,90,30,0)
dist1 =
    75.5225
dist2 = distance(30,-90,30,0)
dist2 =
    75.5225
```

Consider a third point, (`lat`,`lon`), that is also equidistant from the above two points, but at a lesser distance:

```
dist1 = distance(30,90,lat,lon)
dist1 =
    60.0000
dist2 = distance(30,-90,lat,lon)
dist2 =
    60.0000
```

What is this mystery point? The `lat` is 90°N, and any `lon` will do. The North Pole is the true geographic mean of these two points. Note that the great circle containing both points runs through the North Pole (a great circle represents the shortest path between two points on a sphere).

The Mapping Toolbox function `meanm` determines the geographic mean of any number of points. It does this using three-dimensional vector addition of all the points. For example, try the following:

```
lats = [30 30];
longs = [-90 90];
[latbar,longbar] = meanm(lats,longs)
latbar =
    90
longbar =
    0
```

This is the answer you now expect. This geographic mean can result in one oddity; if the vectors all cancel each other, the mean is the center of the planet. In this case, the returned mean point is (`NaN,NaN`) and a warning is displayed. This phenomenon is highly improbable in *real* data, but can be easily constructed. For example, it occurs when all the points are equally spaced along a great circle. Try taking the geographic mean of (0°,0°), (0°,120°), and (0°,240°), which trisect the equator.

```
elats = [0 0 0];
elons = [0 120 240];
meanm(elats,elons)
```

MATLAB issues a warning and returns a vector of `NaN` values.

Geographic Standard Deviation

As you might now expect, the Cartesian definition of standard deviation provided in the standard MATLAB function `std` is also inappropriate for geographic data that is unprojected or covers a significant portion of a planet. Depending upon your purpose, you might want to use the separate geographic deviations for latitude and longitude provided by the function `stdm`, or the single standard distance provided in `stdist`. Both methods measure the deviation of points from the mean position calculated by `meanm`.

The Meaning of stdm

The `stdm` function handles the latitude and longitude deviations separately.

```
[latstd,lonstd] = stdm(lat,lon)
```

The function returns two deviations, one for latitudes and one for longitudes.

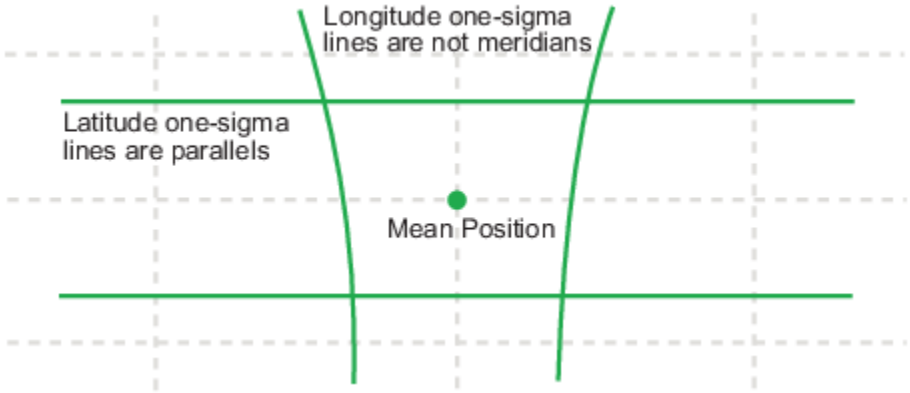
Latitude deviation is a straightforward standard deviation calculation from the mean latitude (mean parallel) returned by `meanm`. This is a reasonable measure for most cases, since on a sphere at least, a degree of latitude always has the same arc length.

Longitude deviation is another matter. Simple calculations based on sum-of-squares angular deviation from the mean longitude (mean meridian) are misleading. The arc length represented by a degree of longitude at extreme latitudes is significantly smaller than that at low latitudes.

The term *departure* is used to represent the arc length distance along a parallel of a point from a given meridian. For example, assuming a spherical planet, the departure of a degree of longitude at the Equator is a degree of arc length, but the departure of a degree of longitude at a latitude of 60° is

one-half a degree of arc length. The `stdm` function calculates a sum-of-squares departure deviation from the mean meridian.

If you want to plot the one-sigma lines for `stdm`, the latitude sigma lines are parallels. However, the longitude sigma lines are not meridians; they are lines of constant departure from the mean parallel.



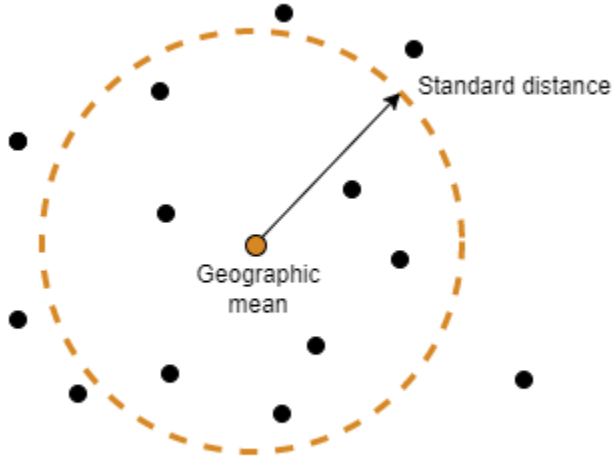
This handling of deviation has its problems. For example, its dependence upon the logic of the coordinate system can cause it to break down near the poles. For this reason, the standard distance provided by `stdist` is often a better measure of deviation. The `stdm` handling is useful for many applications, especially when the data is not global. For instance, these potential difficulties would not be a danger for data points confined to the country of Mexico.

The Meaning of stdist

The standard distance of geographic data is a measure of the dispersion of the data in terms of its distance from the geographic mean. Among its advantages are its applicability anywhere on the globe and its single value:

```
dist = stdist(lat,lon)
```

In short, the standard distance is the average, norm, or *cubic norm* of the distances of the data points in a great circle sense from the mean position. It is probably a superior measure to the two deviations returned by `stdm` except when a particularly latitude- or longitude-dependent feature is under examination.



See Also

More About

- “Equal-Areas in Geographic Statistics” on page 6-36

Equal-Areas in Geographic Statistics

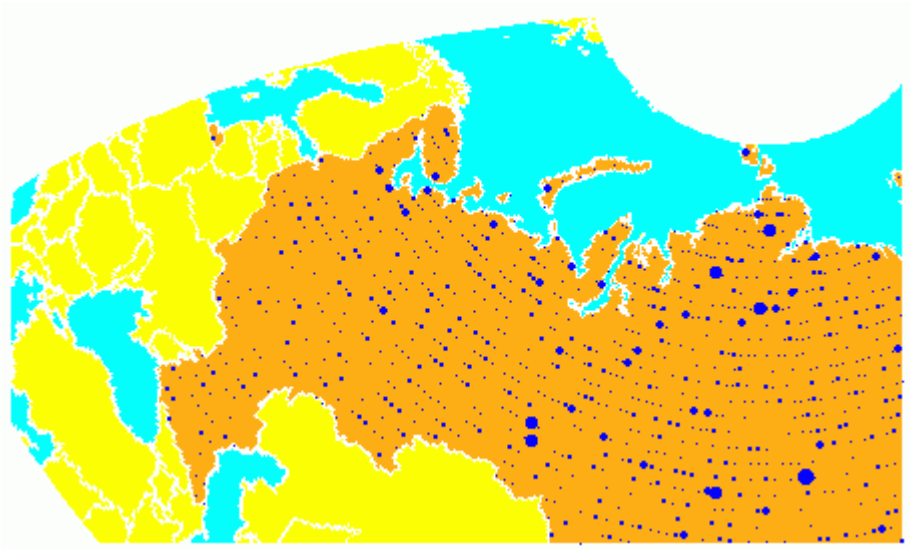
A common error in applying two-dimensional statistics to geographic data lies in ignoring equal-area treatment. It is often necessary to *bin* data to statistically analyze it. In a Cartesian plane, this is easily done by dividing the space into equal *x*-*y* squares. The geographic equivalent of this is to bin up the data in equal latitude-longitude *squares*. Since such squares at high latitudes cover smaller areas than their low-latitude counterparts, the observations in these regions are underemphasized. The result can be conclusions that are biased toward the equator.

Geographic Histograms

The geographic histogram function `histr` allows you to display *binned-up* geographic observations. The `histr` function results in equirectangular binning. Each bin has the same angular measurement in both latitude and longitude, with a default measurement of 1 degree. The center latitudes and longitudes of the bins are returned, as well as the number of observations per bin:

```
[binlat,binlon,num] = histr(lats,lons)
```

As previously noted, these equirectangular bins result in counting bias toward the equator. Here is a display of the one-degree-by-one-degree binning of approximately 5,000 random data points in Russia. The relative size of the circles indicates the number of observations per bin:

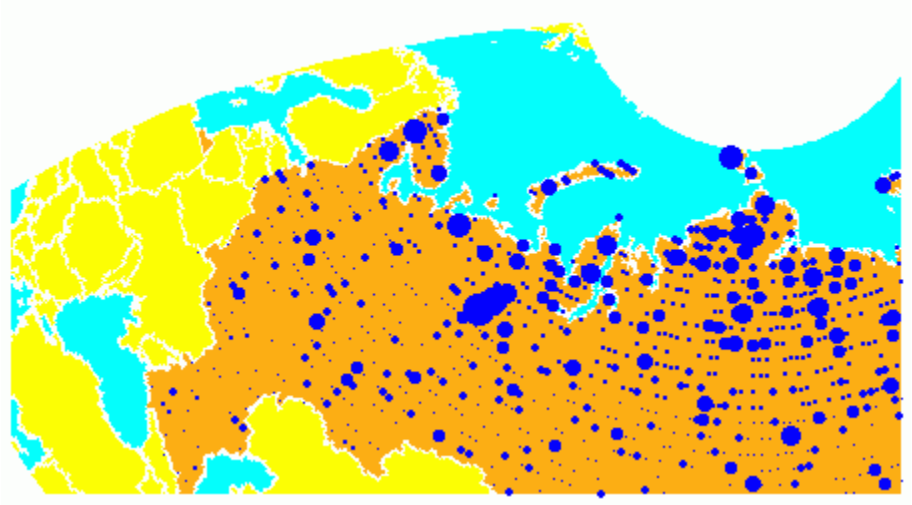


This is a portion of the whole map, displayed in an equal-area Bonne projection. The first step in creating data displays without area bias is to choose an equal-area projection. The proportionally sized symbols are a result of the specialized display function `scatterm`.

You can eliminate the area bias by adding a fourth output argument to `histr`, that will be used to weight each bin's observation by that bin's area:

```
[binlat,binlon,num,wnum] = histr(lats,lons)
```

The fourth output is the weighted observation count. Each bin's observation count is divided by its normalized area. Therefore, a high-latitude bin will have a larger weighted number than a low-latitude bin with the same number of actual observations. The same data and bins look much different when they are area-weighted:



Notice that there are larger symbols to the north in this display. The previous display suggested that the data was relatively uniformly distributed. When equal-area considerations are included, it is clear that the data is skewed to the north. In fact, the data used is northerly skewed, but a simple equirectangular handling failed to demonstrate this.

The `histr` function, therefore, does provide for the display of area-weighted data. However, the actual bins used are of varying areas. Remember, the one-degree-by-one-degree bin near a pole is much smaller than its counterpart near the equator.

The `hista` function provides for actual equal-area bins.

Converting to an Equal-Area Coordinate System

The actual data itself can be converted to an equal-area coordinate system for analysis with other statistical functions. It is easy to convert a collection of geographic latitude-longitude points to an equal-area *x*-*y* Cartesian coordinate system. The `grn2eqa` function applies the same transformation used in calculating the Equal-Area Cylindrical projection:

```
[x,y] = grn2eqa(lat,lon)
```

For each geographic `lat` - `lon` pair, an equal-area `x` - `y` is returned. The variables `x` and `y` can then be operated on under the equal-area assumption, using a variety of two-dimensional statistical techniques. Tools for such analysis can be found in the Statistics and Machine Learning Toolbox™ software and elsewhere. The results can then be converted back to geographic coordinates using the `eqa2grn` function:

```
[lat,lon] = eqa2grn(x, y)
```

Remember, when converting back and forth between systems, latitude corresponds to *y* and longitude corresponds to *x*.

See Also

More About

- “Geographic Statistics for Point Locations on a Sphere” on page 6-32

Georeferenced Raster Data

Raster geodata consists of georeferenced data grids and images that are stored as matrices or objects in the MATLAB workspace. While raster geodata looks like any other matrix of real numbers, what sets it apart is that it is georeferenced, either to the globe or to a specified map projection, so that each pixel of data occupies a known patch of territory on the planet.

All regular data grids require a reference object, matrix, or vector, that specify the placement and resolution of the data set. Geolocated data grids do not require a separate reference object, as they explicitly identify the geographic coordinates of all rows and columns. For details on geolocated grids, see “Geolocated Data Grids” on page 6-53.

Reference Objects

A spatial reference object encapsulates the relationship between a geographic or planar coordinate system and a system of intrinsic coordinates anchored to the columns and rows of a 2-D spatially referenced raster grid or image. A reference object for raster data that is referenced to a geographic latitude-longitude system can be a `GeographicCellsReference` or `GeographicPostingsReference` object. A referencing object for raster data referenced to a planar (projected) map coordinate system can be a `MapCellsReference` or `MapPostingsReference` object. Unlike the older referencing matrix and vector representations (described below), a reference object is self-documenting, providing a rich set of properties to describe both the intrinsic and extrinsic geometry. The use of reference objects is preferred, but some referencing matrix and vector functionality continues to be supported for the purpose of compatibility.

Referencing Matrices

A referencing matrix is a 3-by-2 matrix of doubles that describes the scaling, orientation, and placement of the data grid on the globe. For a given referencing matrix, `R`, one of the following relations holds between rows and columns and coordinates (depending on whether the grid is based on map coordinates or geographic coordinates, respectively):

```
[x y] = [row col 1] * R, or
[long lat] = [row col 1] * R
```

Convert a referencing matrix to a raster reference object using the `refmatToGeoRasterReference` or `refmatToMapRasterReference` functions.

Referencing Vectors

In many instances (when the data grid or image is based on latitude and longitude and is aligned with the geographic graticule), a referencing matrix has more degrees of freedom than the data requires. In such cases, you may encounter a more compact representation, a three-element *referencing vector*. A referencing vector defines the pixel size and northwest origin for a regular, rectangular data grid:

```
refvec = [cells-per-degree north-lat west-lon]
```

This variable is often called `refvec` (or `maplegend`). The first element, cells-per-degree, describes the angular extent of each grid cell (e.g., if each cell covers five degrees of latitude and longitude, cells-per-degree would be specified as `0.2`). Note that if the latitude extent of cells differs from their longitude extent you cannot use a referencing vector, and instead must specify a referencing object or

matrix. The second element, north-lat, specifies the northern limit of the data grid (as a latitude), and the third element, west-lon, specifies the western extent of the data grid (as a longitude). In other words, north-lat, west-lon is the northwest corner of the data grid. Note, however, that cell (1,1) is always in the southwest corner of the grid. This need not be the case for grids or images described by referencing objects or matrices.

Convert a referencing vector to a geographic raster reference object using the `refvecToGeoRasterReference` function.

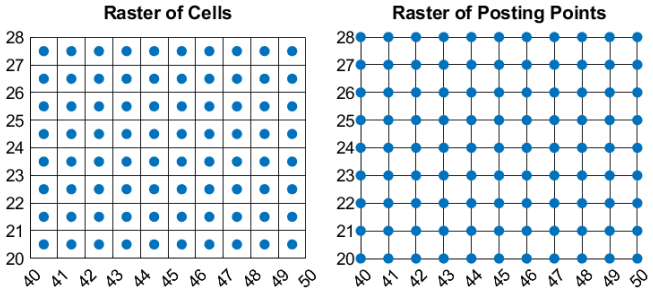
Spatially Reference Imported Rasters

To associate the elements of a raster with geospatial locations, the raster must be accompanied by spatial referencing information such as the geographic or world limits, the distance between recorded samples, and the extent of individual cells. When a supported raster data file contains spatial referencing information, you can use the `readgeoraster` function to import the data as an array and the referencing information as a raster reference object. If the data file does not contain referencing information, then you can import the data as an array and create a reference object using the `georefcells`, `georefpostings`, `maprefcells`, or `maprefpostings` function. The creation function you use depends on characteristics of the raster data.

- `georefcells` — The raster is a grid of quadrangular cells referenced to geographic latitude and longitude coordinates.
- `georefpostings` — The raster is a grid of posting point samples referenced to geographic coordinates.
- `maprefcells` — The raster is a grid of rectangular cells referenced to projected world *x*- and *y*-coordinates.
- `maprefpostings` — The raster is a grid of posting point samples referenced to projected coordinates.

Differentiate Between Cells and Postings

This image shows differences between a projected raster of cells and a projected raster of posting points. Both rasters have elements that are spaced 1 meter apart, with the raster covering *x*-values in the range [40,50] in meters and *y*-values in the range [20,28] in meters. The raster of cells is 8-by-10 and the raster of postings is 9-by-11. The boundary of the raster of cells is made up of the outermost boundaries of the outermost cells and the boundary of the raster of postings is made up of sampling points along the edges of the raster.



If you do not know whether your raster is a grid of cells or a grid of posting points, you can try the following:

- Ask your data provider.
- Search the metadata for information about the spatial registration or interpretation of the data. Metadata for rasters of cells can contain phrases such as "pixels" or "pixel is area." Metadata for rasters of posting points can contain phrases such as "grid", "node", or "pixel is point."
- Consider what the data represents. Images are typically made of cells, while elevation grids are typically made of posting points.
- Consider the size of the raster. If the dimensions of the raster are round numbers, such as a raster of size `[1000 1000]`, then the raster is probably made of cells. If the dimensions of the raster are

round numbers plus one, such as a raster of size `[1001 1001]`, then the raster is probably made of posting points.

Spatially Reference an Image

This example shows how to import an image, spatially reference the image by creating a reference object, then display the image on a map.

Import an image as an array by using the `imread` function. The array is of size 500-by-500-by-3 and specifies the red, green, and blue components of the image.

```
A = imread('boston_common.jpg');
```

To spatially reference the image, you must determine the following:

- Whether the image is referenced to geographic or projected coordinates
- Whether the image is made up of cells or posting points

Information included in the file `boston_common.txt` indicates that the image is referenced to projected coordinates and is made up of cells. Therefore, you can create a reference object by using the `maprefcells` function. Specify the *x*- and *y*-limits, also included in the file `boston_common.txt`, using world coordinates.

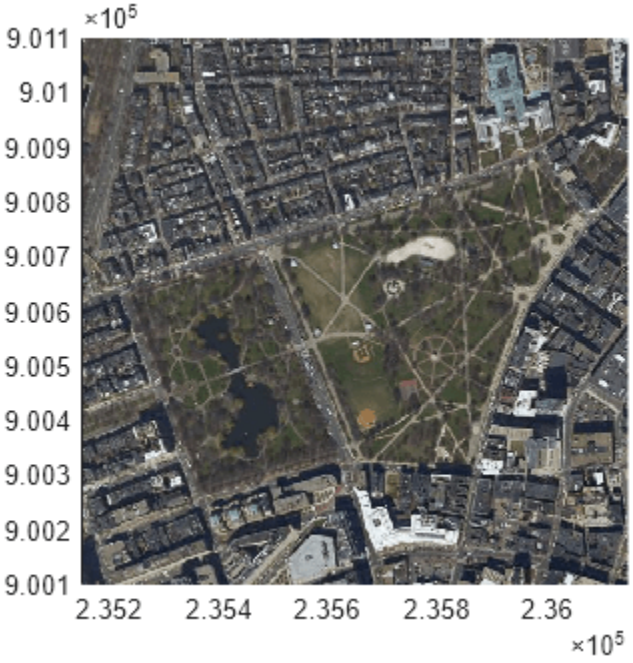
```
xlimits = [235150 236150];
ylimits = [900100 901100];
R = maprefcells(xlimits,ylimits,size(A));
```

Define the first row of `A` as the northernmost edge of the image by setting the `ColumnsStartFrom` property of the reference object to `'north'`. Otherwise, the `ColumnsStartFrom` property defaults to `'south'`.

```
R.ColumnsStartFrom = 'north';
```

Display the spatially referenced image on a map by using the `mapshow` function.

```
mapshow(A,R)
```



The data used in this example is derived from data provided by MassGIS (Bureau of Geographic Information). See the file `boston_common.txt` for more details.

Spatially Reference an Elevation Grid

This example shows how to import elevation data, spatially reference the data by creating a reference object, then display the data on an `axesm`-based map.

Load elevation data as an array.

```
load elevation_n39_w106.mat
```

To spatially reference the data, you must determine the following:

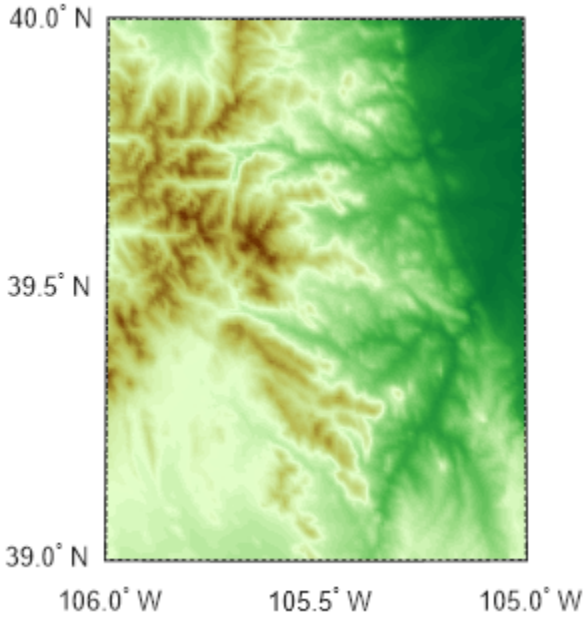
- Whether the data is referenced to geographic or projected coordinates
- Whether the data is made of cells or posting points

Information in the file `elevation_n39_w106.txt` indicates that the data set is referenced to geographic coordinates and is made up of posting points. Therefore, you can create a reference object for the data by using the `georefpostings` function. Specify the latitude and longitude limits, also included in the file `boston_common.txt`, using degrees.

```
latlim = [39 40];
lonlim = [-106 -105];
R = georefpostings(latlim,lonlim,size(elevation_n39_w106));
```

Create an `axesm`-based map by specifying the latitude and longitude limits of the data. Then, display the data as a surface by using the `geoshow` function. Apply a colormap appropriate for elevation data by using the `demcmap` function.

```
usamap(R.LatitudeLimits,R.LongitudeLimits)
geoshow(elevation_n39_w106,R,'DisplayType','surface')
demcmap(elevation_n39_w106)
```



The data set used in this example is derived from data provided by the U.S. Geological Survey. See the file `elevation_n39_w106.txt` for more details.

See Also

Functions
`readgeoraster` | `georefcells` | `georefpostings` | `maprefcells` | `maprefpostings`

Objects
`MapCellsReference` | `GeographicPostingsReference` | `GeographicCellsReference` | `MapPostingsReference`

Merge Multiple Raster Tiles Using Datastore

This topic shows how to:

- Create a custom datastore for reading and merging data from multiple DTED files.
- Use the custom datastore to read and merge data from multiple DTED files.
- Adapt the custom datastore to GTOPO30 and GLOBE DEM files.

A datastore is a repository for a collection of files. Datastores enable you to read data from all files in the collection without specifying the names of individual files, which is convenient when the collection contains a large number of files. Custom datastores enable you to define additional functions that manipulate the data. In this case, the datastore defines a function that simultaneously reads and merges the data from the files.

This topic assumes familiarity with datastores and with creating classes. For more information about datastores, see “Getting Started with Datastore”. For more information about creating classes, see “Creating a Simple Class”.

Create Custom Datastore

Create a custom datastore, `DTEDDatastore`, for reading and merging DTED files. The steps for creating a custom datastore include:

- Creating a class definition file that defines the custom datastore.
- Inheriting from the base datastore class.
- Implementing a method that creates datastore objects.
- Implementing methods that read and merge data from files in the datastore.

The “Merge DTED Tiles Using Datastore” on page 6-48 example at the end of this topic includes the completed `DTEDDatastore` class as a supporting file.

Create Class Definition File

Create a class definition in an `.m` file called `DTEDDatastore.m`. This file defines the `DTEDDatastore` class, including its properties and methods. The file must be on the MATLAB path.

Inherit from Base Datastore Class

Within the file, write code that inherits from the base `matlab.io.Datastore` class. Inheriting from the base `matlab.io.Datastore` class enables you to use methods that are common to all datastore objects, without explicitly defining the methods. For example, you can use the `readall` method to read all the files contained in a DTED datastore, even though the `DTEDDatastore` class does not define a `readall` method.

```
classdef DTEDDatastore < matlab.io.Datastore
% DTEDDatastore Datastore for a collection of DTED files
```

Add Properties to Class

Write code that adds the `CurrentFileIndex` and `FileSet` properties to the class.

- `CurrentFileIndex` stores the index of the next file to be read by the datastore. The `read`, `reset`, and `progress` methods use this property.

- `FileSet` stores information about the files in the datastore using a `DsFileSet` object. The `hasdata`, `read`, `reset`, and `progress` methods use this property.

```
properties (Access = private)
    CurrentFileIndex double
    FileSet matlab.io.datastore.DsFileSet
end
```

Implement Creation Function

Implement a method that creates `DTEDDatastore` objects.

Write code that defines the `DTEDDatastore` method. The method creates a datastore for a collection of DTED files. The files exist in the folder specified by `location` and have the extension specified by `extension`. The function sets the `FileSet` property of the datastore to a `DsFileSet` object that stores the extension and enables the datastore to access subfolders of `location`. The function also resets the datastore to an initial state.

```
methods

    function dtedDS = DTEDDatastore(location,extension)
        % Create a DTEDDatastore object
        arguments
            location {mustBeFolder}
            extension {mustBeTextScalar}
        end

        dtedDS.FileSet = matlab.io.datastore.DsFileSet(location, ...
            FileExtensions=extension,IncludeSubfolders=true);

        reset(dtedDS)
    end
```

Implement Methods

Custom datastores that use a `DsFileSet` object require you to define the `hasdata`, `read`, `reset`, `progress`, and `copyElement` methods. You must also define a method that reads and merges data from all files in the datastore.

Implement the `hasdata` method, which returns logical `1` (`true`) if there is data available to read from the datastore. Otherwise, it returns logical `0` (`false`).

```
    function tf = hasdata(dtedDS)
        % Return true if there is data available to read in the
        % datastore
        tf = hasfile(dtedDS.FileSet);
    end
```

Implement the `read` method, which reads data and information from the next file in the datastore. The function returns the data in `data` and returns information about the file in `info`.

Within the method:

- If there is no data to read, issue an error.
- Get the name of the next file in the datastore.
- Read data from the file by using the `readgeoraster` function. The function returns an array that contains the elevation data and a raster reference object that contains spatial referencing

information. To use the elevation data with other Mapping Toolbox functions, specify the output type as `"double"`.

- Get information about the file by using the `georasterinfo` function. Store the information in the `info` output argument.
- Collect the array and the raster reference object into a cell array. Store the cell array in the `data` output argument.
- Increment the index of the next file to be read.

```
function [data,info] = read(dtedDS)
    % Read data and information from the next file in the datastore

    % If there is no data to read, issue error
    if ~hasdata(dtedDS)
        error(sprintf("No more data to read. Reset datastore " + ...
            "to initial state by using reset."))
    end

    % Get name of next file
    fileInfo = nextfile(dtedDS.FileSet);
    filename = fileInfo.FileName;

    % Read file
    [Z,R] = readgeoraster(filename,OutputType="double");

    % Get information about file
    info = georasterinfo(filename);

    % Collect array and raster reference object into cell array
    data = {Z,R};

    % Increment index of next file to read
    dtedDS.CurrentFileIndex = dtedDS.CurrentFileIndex + 1;
end
```

Implement the `reset` method, which resets the datastore to the state where no data has been read from it. Resetting enables you to re-read from the same datastore.

```
function reset(dtedDS)
    % Reset datastore to initial state
    reset(dtedDS.FileSet)
    dtedDS.CurrentFileIndex = 1;
end
```

Implement the `readMergedTiles` method, which reads data from all the files in the datastore, and then merges the data into one array and one raster reference object.

Within the method:

- Read data from all the files in the datastore by using the `readall` function. The `readall` function returns a two-column cell array. The first column contains the elevation data and the second column contains the raster reference objects.
- Reshape the data into a one-row cell array that you can use as input with the `mergetiles` function.
- Extract the data from the cell array, and merge the data by using the `mergetiles` function.

```
        function [Z,R] = readMergedTiles(dtedDS)
            % Read data from all files and merge into one array and one
            % raster reference object

            % Read all data
            alldata = readall(dtedDS)';

            % Reshape data
            alldata = alldata(:)';

            % Extract data from cell array and merge data
            [Z,R] = mergetiles(alldata{:});
        end

    end
```

Implement the `progress` method, which determines the percentage of data that has been read from the datastore.

```
    methods (Hidden = true)
        function frac = progress(dtedDS)
            % Determine percentage of data read from datastore
            if hasdata(dtedDS)
                frac = (dtedDS.CurrentFileIndex-1)/dtedDS.FileSet.NumFiles;
            else
                frac = 1;
            end
        end
    end
```

Implement the `copyElement` method, which you must use when you store a `DsFileSet` object within a property of the datastore. Implementing the `copyElement` method enables the class to create a deep copy of the datastore object.

```
    methods (Access = protected)
        % Enable class to create deep copy of datastore object
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end
    end
end
```

Merge DTED Tiles Using Datastore

Read and merge multiple DTED files by using a custom datastore. A supporting file for the example, `DTEDDatastore.m`, defines a custom datastore for reading and merging DTED files. To read and merge files using the custom datastore, the DTED files must form a filled quadrangle. A *quadrangle* is a region bounded by two parallels and two meridians.

Create DTED Datastore Object

Specify the location of the folder that contains the DTED files, in this case the current folder. Two supporting folders for the example, `w106` and `w107`, contain several DTED files.

```
location = pwd;
```

Specify the extension of the DTED files. Valid extensions for DTED files are `.dt0`, `.dt1`, and `.dt2`.

```
extension = ".dt0";
```

Create a `DTEDDatastore` object by specifying the location and extension. The amount of time MATLAB® requires to create the object depends on the number of subfolders.

```
dtedDS = DTEDDatastore(location,extension);
```

Read and Merge Data from Files

Read and merge the files in the DTED datastore by using the `readMergedTiles` function. The `DTEDDatastore.m` file defines the `readMergedTiles` function for `DTEDDatastore` objects.

```
[Z,R] = readMergedTiles(dtedDS);
```

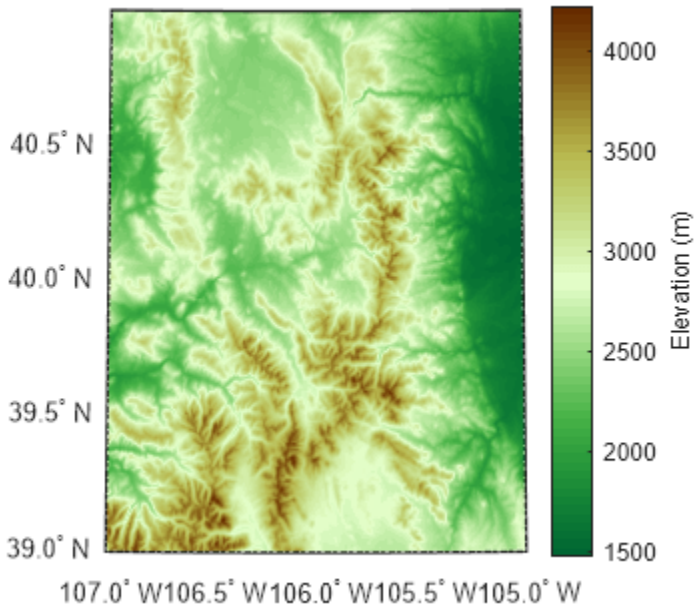
Display Data on Map

Create a map for the region and display the elevation data as a surface.

```
figure
usamap(Z,R)
geoshow(Z,R,DisplayType="surface")
```

Apply a colormap that is appropriate for elevation data by using the `demcmap` function. Add a labeled color bar.

```
demcmap(Z)

c = colorbar;
c.Label.String = "Elevation (m)";
```



This example uses modified data from the US Geological Survey.

Adapt DTED Datastore for Other Data Sets

You can use a custom datastore that is similar to the DTED datastore to read and merge other tiled data, such as data from GTOPO30 or GLOBE DEM files. Depending on the data set, you might encounter limitations or need to change the implementation of the `read` method.

For GTOPO30 files:

- The `readMergedTiles` method requires the data to fill a quadrangle, but the data stored in the southernmost GTOPO30 tiles have different longitude limits than the tiles that are adjacent to the north. As a result, this custom datastore does not support collections that mix southernmost tiles with other tiles.
- When you create the datastore object, specify the extension as `".dem"`.

For GLOBE DEM files:

- When you implement the `read` method, you must update the use of the `readgeoraster` function so that the function specifies a geographic coordinate system type, as in `CoordinateSystemType="geographic"`.
- When you implement the `read` method, you can suppress and reenable warnings from the `georasterinfo` function by adding this code.

  ```
  warning("off","map:io:UnableToDetermineCoordinateSystemType")
  info = georasterinfo(filename);
  warning("on","map:io:UnableToDetermineCoordinateSystemType")
  ```
- When you create the datastore object, specify the extension as `""`.

See Also

Functions
`readgeoraster` | `mergetiles`

Classes
`matlab.io.Datastore` | `matlab.io.datastore.DsFileSet`

Related Examples

- “Develop Custom Datastore”

Construct a Global Data Grid

Imagine an extremely coarse map of the world in which each cell represents 60°. Such a map matrix would be 3-by-6.

1. Create a 3-by-6 grid:

```
miniZ = [1 2 3 4 5 6; 7 8 9 10 11 12; 13 14 15 16 17 18];
```

2. Now make a referencing object:

```
miniR = georasterref('RasterSize', size(miniZ), ...
   'Latlim', [-90 90], 'Lonlim', [-180 180])
```

   Your output appears like this:

```
miniR =

  GeographicCellsReference with properties:

            LatitudeLimits: [-90 90]
           LongitudeLimits: [-180 180]
                RasterSize: [3 6]
      RasterInterpretation: 'cells'
          ColumnsStartFrom: 'south'
             RowsStartFrom: 'west'
      CellExtentInLatitude: 60
     CellExtentInLongitude: 60
    RasterExtentInLatitude: 180
   RasterExtentInLongitude: 360
         XIntrinsicLimits: [0.5 6.5]
         YIntrinsicLimits: [0.5 3.5]
     CoordinateSystemType: 'geographic'
                AngleUnit: 'degree'
```

3. Set up an equidistant cylindrical map projection:

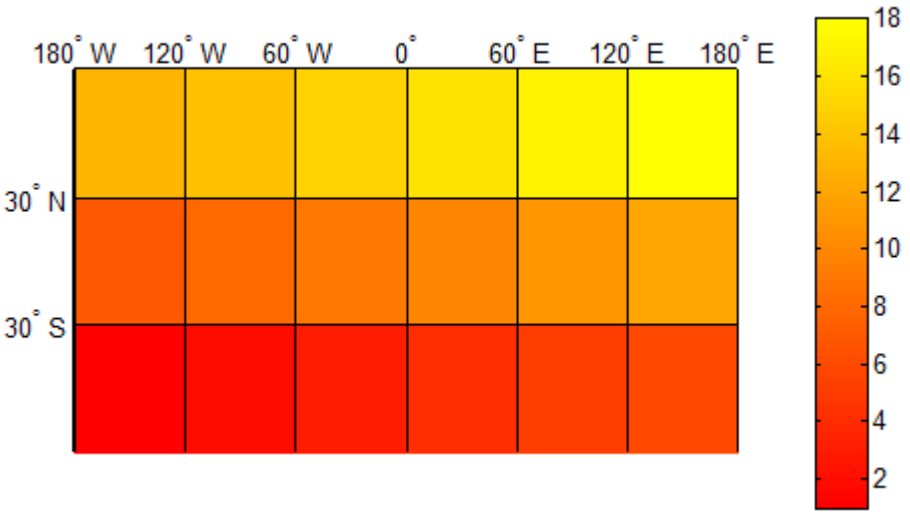
```
figure('Color','white')
ax = axesm('MapProjection', 'eqdcylin');
axis off
setm(ax,'GLineStyle','-', 'Grid','on','Frame','on')
```

4. Draw a graticule with parallel and meridian labels at 60° intervals:

```
setm(ax, 'MlabelLocation', 60, 'PlabelLocation',[-30 30],...
    'MLabelParallel','north', 'MeridianLabel','on',...
    'ParallelLabel','on','MlineLocation',60,...
    'PlineLocation',[-30 30])
```

5. Map the data using `geoshow` and display with a color ramp and legend:

```
geoshow(miniZ, miniR, 'DisplayType', 'texturemap');
colormap('autumn')
colorbar
```



Note that the first row of the matrix is displayed at the bottom of the map, while the last row is displayed at the top.

Geolocated Data Grids

In addition to regular data grids, the toolbox provides another format for geodata: geolocated data grids. These multivariate data sets can be displayed, and their values and coordinates can be queried, but unfortunately much of the functionality supporting regular data grids is not available for geolocated data grids.

Regular data grids cover simple, regular quadrangles, that is, geographically rectangular and aligned with parallels and meridians. Geolocated data grids, in addition to these rectangular orientations, can have other shapes as well.

Define Geolocated Data Grid

To define a geolocated data grid, you must define three variables: a matrix of indices or values associated with the mapped region, a matrix giving cell-by-cell latitude coordinates, and a matrix giving cell-by-cell longitude coordinates.

Load a MAT-file containing an irregularly shaped geolocated data grid called `mapmtx`.

```
load mapmtx
```

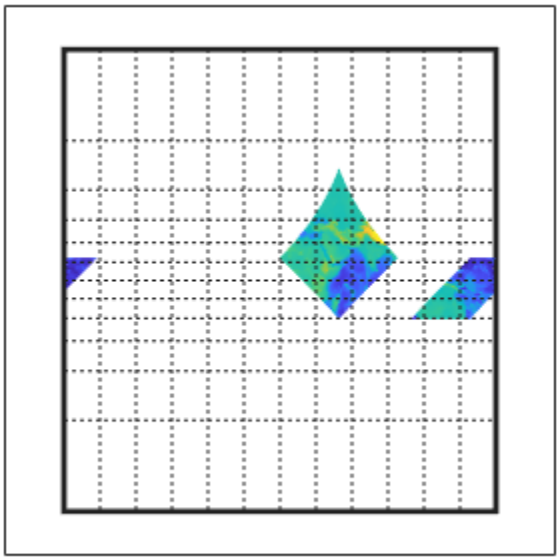
View the variables created from this MAT-file. Two geolocated data grids are in this data set, each requiring three variables. The values contained in `map1` correspond to the latitude and longitude coordinates, respectively, in `lt1` and `lg1`. Notice that all three matrices are the same size. Similarly, `map2`, `lt2`, and `lg2` together form a second geolocated data grid. These data sets were extracted from the `topo60c` data grid. Neither of these maps is regular, because their columns do not run north to south.

```
whos
```

```
  Name              Size            Bytes  Class     Attributes

  description       1x54              108  char
  lg1              50x50            20000  double
  lg2              50x50            20000  double
  lt1              50x50            20000  double
  lt2              50x50            20000  double
  map1             50x50            20000  double
  map2             50x50            20000  double
  source            1x43               86  char
```

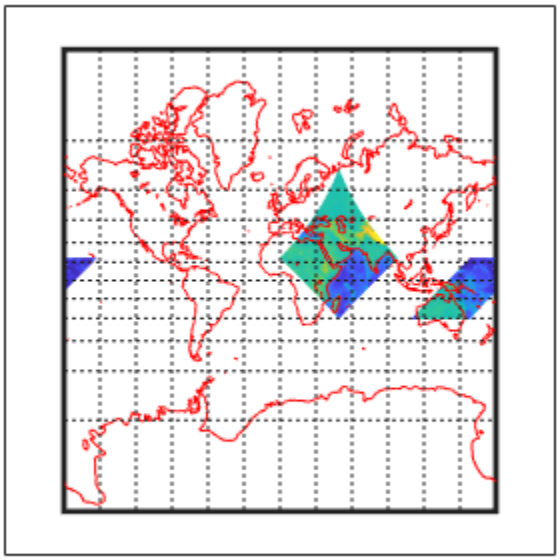
Display the grids one after another to see their geography.

```
close all
axesm mercator
gridm on
framem on
h1 = surfm(lt1,lg1,map1);
h2 = surfm(lt2,lg2,map2);
```



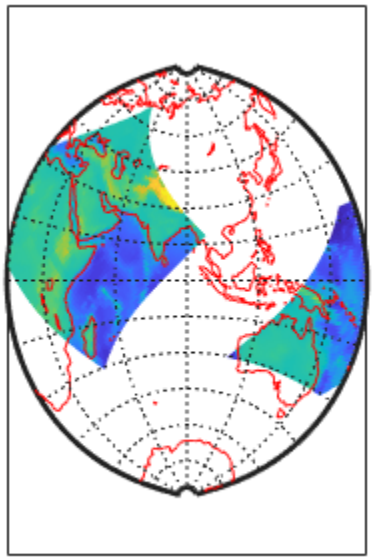
Showing coastlines will help to orient you to these skewed grids. Notice that neither grid is a regular rectangle. One looks like a diamond geographically, the other like a trapezoid. The trapezoid is displayed in two pieces because it crosses the edge of the map. These shapes can be thought of as the geographic organization of the data, just as rectangles are for regular data grids. But, just as for regular data grids, this organizational logic does not mean that displays of these maps are necessarily a specific shape.

```
load coastlines
plotm(coastlat,coastlon,'r')
```



Now change the view to a polyconic projection with an origin at 0°N, 90°E. As the polyconic projection is limited to a 150° range in longitude, those portions of the maps outside this region are automatically trimmed.

```
setm(gca,'MapProjection','polycon','Origin',[0 90 0])
```



Geographic Interpretations of Geolocated Grids

Mapping Toolbox software supports three different interpretations of geolocated data grids:

- First, a map matrix having the same number of rows and columns as the latitude and longitude coordinate matrices represents the values of the map data at the corresponding geographic points (centers of data cells).
- Next, a map matrix having one fewer row and one fewer column than the geographic coordinate matrices represents the values of the map data within the area formed by the four adjacent latitudes and longitudes.
- Finally, if the latitude and longitude matrices have smaller dimensions than the map matrix, you can interpret them as describing a coarser *graticule*, or mesh of latitude and longitude cells, into which the blocks of map data are warped.

This section discusses the first two interpretations of geolocated data grids. For more information on the use of graticules, see “The Map Grid” on page 4-39.

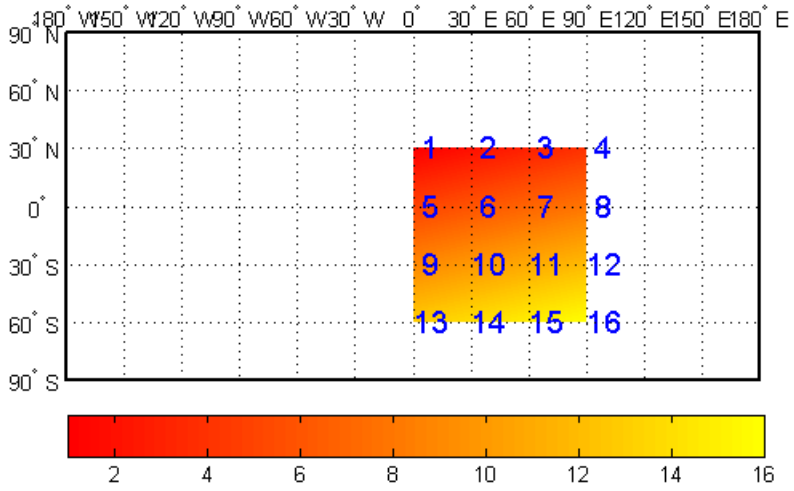
Type 1: Values Associated with the Upper Left Grid Coordinate

As an example of the first interpretation, consider a 4-by-4 map matrix whose cell size is 30-by-30 degrees, along with its corresponding 4-by-4 latitude and longitude matrices:

```
Z = [ ...
 1  2  3  4; ...
 5  6  7  8; ...
 9 10 11 12; ...
13 14 15 16];

lat = [ ...
  30  30  30  30; ...
   0   0   0   0; ...
 -30 -30 -30 -30; ...
 -60 -60 -60 -60];

lon = [ ...
 0 30 60 90;...
 0 30 60 90;...
 0 30 60 90;...
 0 30 60 90];
```

Display the geolocated data grid with the values of `map` shown at the associated latitudes and longitudes:

```
figure('Color','white','Colormap',autumn(64))
axesm('pcarree','Grid','on','Frame','on',...
    'PLineLocation',30,'PLabelLocation',30)
mlabel; plabel; axis off; tightmap

h = geoshow(lat,lon,Z,'DisplayType','surface');
set(h,'ZData',zeros(size(Z)))
ht = textm(lat(:),lon(:),num2str(Z(:)), ...
    'Color','blue','FontSize',14);

colorbar('southoutside')
```



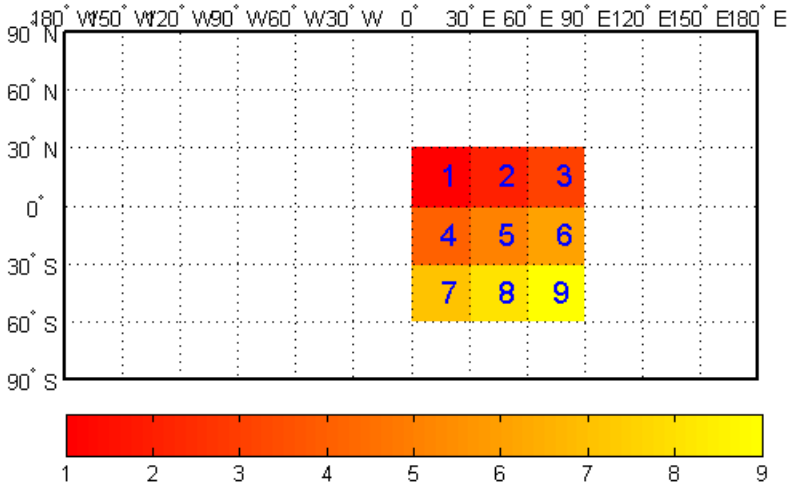
Notice that only 9 of the 16 total cells are displayed. The value displayed for each cell is the value at the upper left corner of that cell, whose coordinates are given by the corresponding `lat` and `lon` elements. By convention, the last row and column of the map matrix are not displayed, although they exist in the `CData` property of the surface object.

Type 2: Values Centered Within Four Adjacent Coordinates

For the second interpretation, consider a 3-by-3 map matrix with the same `lat` and `lon` variables:

```
delete(h)
delete(ht)

Z3by3 = [ ...
 1  2  3; ...
 4  5  6; ...
 7  8  9];

h = geoshow(lat,lon,Z3by3,'DisplayType','texturemap');

tlat = [ ...
   15 15 15; ...
   -15 -15 -15; ...
   -45 -45 -45];

tlon = [ ...
    15 45 75; ...
    15 45 75; ...
    15 45 74];

textm(tlat(:),tlon(:),num2str(Z3by3(:)), ...
    'Color','blue','FontSize',14)
```

Display a surface plot of the map matrix, with the values of `map` shown at the center of the associated cells:



All the map data is displayed for this geolocated data grid. The value of each cell is the value at the center of the cell, and the latitudes and longitudes in the coordinate matrices are the boundaries for the cells.

Ordering of Cells

You may have noticed that the first row of the matrix is displayed as the top of the map, whereas for a regular data grid, the opposite was true: the first row corresponded to the bottom of the map. This difference is entirely due to how the `lat` and `lon` matrices are ordered. In a geolocated data grid, the order of values in the two coordinate matrices determines the arrangement of the displayed values.

Transform Regular to Geolocated Grids

When required, you can transform a regular data grid into a geolocated data grid by using the `geographicGrid` function.

Transforming Geolocated to Regular Grids

Conversely, a regular data grid can also be constructed from a geolocated data grid. The coordinates and values can be embedded in a new regular data grid. The function that performs this conversion is `geoloc2grid`; it takes a geolocated data grid and a cell size as inputs.

Data Grids as Logical Variables

You can apply logical criteria to numeric data grids to create *logical grids*. Logical grids are data grids consisting entirely of 1s and 0s. You can create them by performing logical tests on data grid variables. The resulting binary grid is the same size as the original grid(s) and can use the same referencing vector, as the following hypothetical data operation illustrates:

```
logicalgrid = (realgrid > 0);
```

This transforms all values greater than 0 into 1s and all other values to 0s. You can apply multiple conditions to a grid in one operation:

```
logicalgrid = (realgrid >- 100)&(realgrid < 100);
```

If several grids are the same size and share the same referencing vector (i.e., the grids are co-registered), you can create a logical grid by testing joint conditions, treating the individual data grids as map layers:

```
logicalgrid = (population > 10000)&(elevation < 400)&...
              (country == nigeria);
```

Several Mapping Toolbox functions enable the creation of logical grids using logical and relational operators. Grids resulting from such operations contain logical rather than numeric values (which reduce storage by a factor of 8), but might need to be cast to `double` in order to be used in certain functions. Use the `ones` and `zeros` functions to create grids of all 1s and all 0s.

Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid

This example shows how to combine an elevation data grid and an attribute (color) data grid that cover the same region but are gridded differently. The example drapes slope data from a regular data grid on top of elevation data from a geolocated data grid. The example uses the geolocated data grid as the source for surface elevations and transforms the regular data grid into slope values, which are then sampled to conform to the geolocated data grid (creating a set of slope values for the diamond-shaped grid) and color-coded for surface display. This approach works with any dissimilar grids, although the two data sets in this example actually have the same origin (the geolocated grid derives from the `topo60c` data set).

Load the geolocated data grid from the `mapmtx` file and the regular data grid from the `topo60c` file. The `mapmtx` file actually contains two regions but this example only uses the diamond-shaped portion, `lt1`, `lg1`, and `map1`, centered on the Middle East.

```
load mapmtx lt1 lg1 map1
load topo60c
```

Compute the surface aspect, slope, and gradients for `topo60c`. This example only uses the slopes in subsequent steps.

```
[aspect,slope,gradN,gradE] = gradientm(topo60c,topo60cR);
```

Use the `geointerp` function to interpolate slope values to the geolocated grid specified by `lt1` and `lg1`. The output is a 50-by-50 grid of elevations matching the coverage of the `map1` variable.

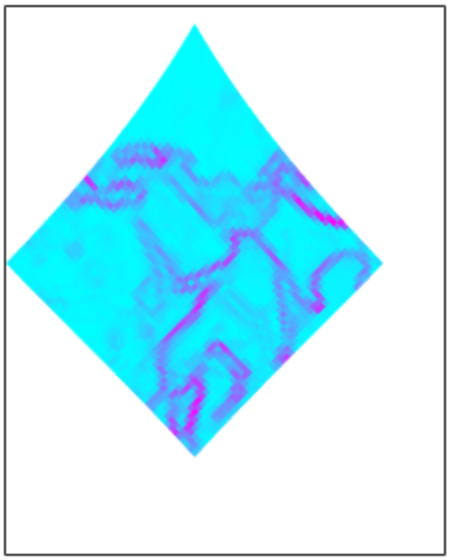
```
slope1 = geointerp(slope,topo60cR,lt1,lg1);
```

Set up a figure with a Miller projection and use `surfm` to display the slope data. Specify the *z* -values for the surface explicitly as the `map1` data, which is terrain elevation. The map mainly depicts steep cliffs, which represent mountains (the Himalayas in the northeast), and continental shelves and trenches.

```
figure
axesm miller
surfm(lt1,lg1,slope1,map1)
```

The coloration depicts steepness of slope. Change the colormap to make the steepest slopes magenta, the gentler slopes dark blue, and the flat areas light blue:

```
colormap cool
```



Use `view` to get a southeast perspective of the surface from a low viewpoint. In 3-D, you can see the topography as well as the slope.

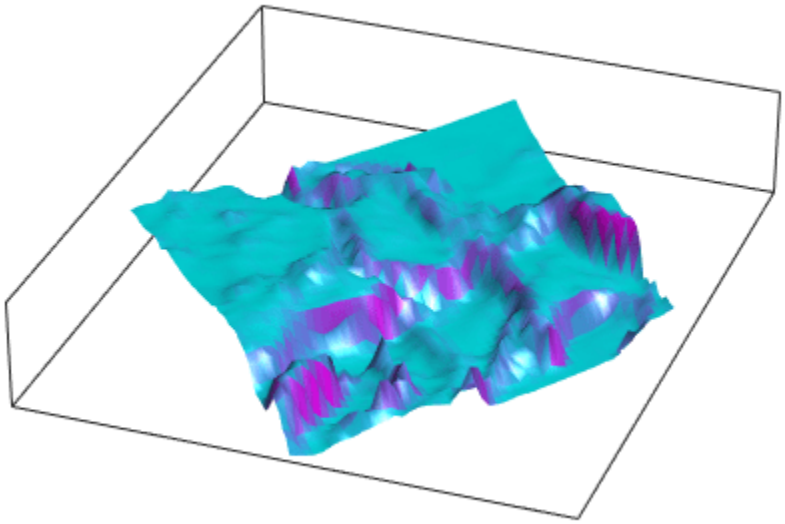
```
view(20,30)
daspectm('meter',200)
```

The default rendering uses faceted shading (no smooth interpolation). Make the surface shiny with Gouraud shading and specify lighting from the east (the default of `camlight` lights surfaces from over the right shoulder of the viewer).

```
material shiny
camlight
lighting Gouraud
```

Remove the white space and view the figure in perspective mode.

```
axis tight
ax = gca;
ax.Projection = 'perspective';
```



Georeference an Image to an Orthotile Base Layer

This example shows how to register an image to an Earth-based coordinate reference system (CRS) and create a new georeferenced image.

The steps of the example include:

- Read orthophoto base tiles that are georeferenced to map coordinates.
- Read an aerial photograph that is not georeferenced.
- Register the aerial photograph to map coordinates.
- Use the registration to apply a geometric transformation to the aerial photograph.
- Display the aerial photograph in map coordinates.
- Overlay a vector layer of road data.

All georeferenced data in the example uses the Massachusetts State Plane projected CRS. The projected CRS defines the map coordinates. The reference ellipsoid underlying the projected CRS is the North American Datum of 1983 in units of meters.

Read and Display Orthophoto Base Tiles

Read two orthophoto base tiles for West Concord, Massachusetts into the workspace. Create reference objects for the orthophotos by reading their world files.

```
[baseImage1,cmap1] = imread("concord_ortho_w.tif");
[baseImage2,cmap2] = imread("concord_ortho_e.tif");

R1 = worldfileread("concord_ortho_w.tfw","planar",size(baseImage1));
R2 = worldfileread("concord_ortho_e.tfw","planar",size(baseImage2));
```

Merge the tiles into one image.

```
[baseImage,R] = mergetiles(baseImage1,R1,baseImage2,R2);
```

Verify that the colormaps for the images are equal.

```
isequal(cmap1,cmap2)
```

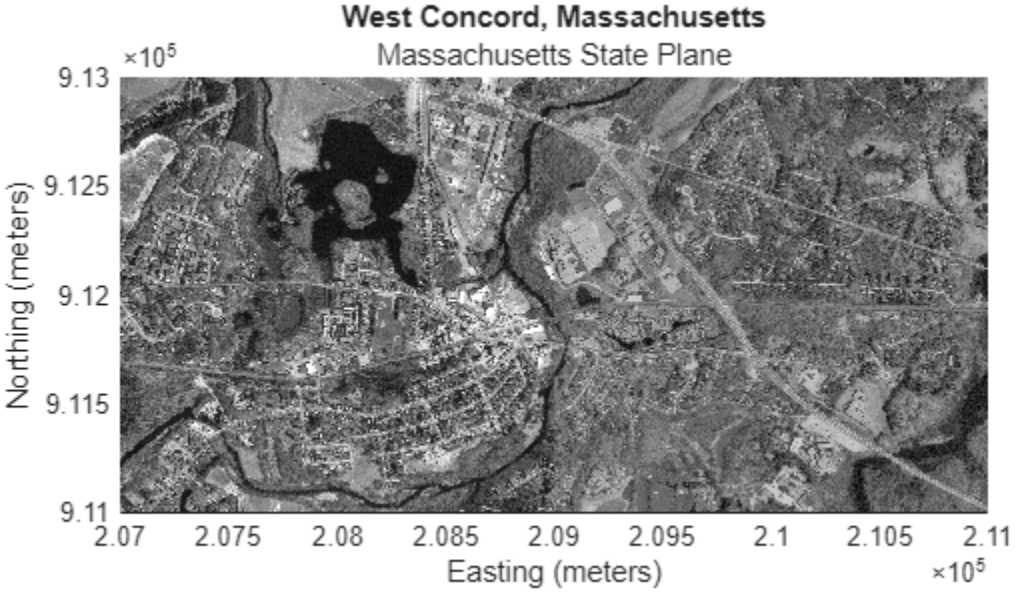
```
ans = logical
   1
```

Generalize the name of the colormap variable `cmap1` to `cmap`.

```
cmap = cmap1;
```

Display the merged image on a map. Add a title, subtitle, and axes labels.

```
figure
mapshow(baseImage,cmap,R)

title("West Concord, Massachusetts")
subtitle("Massachusetts State Plane")

xlabel("Easting (meters)")
ylabel("Northing (meters)")
```



Store the current axes in a variable for future use.

```
ax1 = gca;
```

Read and Display Aerial Photograph

Read and display an aerial photograph of West Concord, Massachusetts. The orthophoto base tiles cover the extent of the photograph.

```
inputImage = imread("concord_aerial_sw.jpg");

figure
imshow(inputImage)
title("Unregistered Aerial Photograph")
```

Register Aerial Photograph to Map Coordinates

Register the aerial photograph to map coordinates by selecting control point pairs that relate the aerial photograph to the orthophoto base layer. Control points are landmarks that you can identify in both images, such as road intersections or natural features.

The merged orthophoto image is an indexed image. Prepare to select control points by converting the indexed image to a grayscale image.

```
baseGray = im2uint8(ind2gray(baseImage,cmap));
```

Load six preselected control point pairs from the file `cpstruct.mat`.

```
load cpstruct.mat
```

Optionally, edit the preselected points or add additional points by using the Control Point Selection tool. Before opening the tool, downsample the images by a factor of 2. Save the control points by selecting the **File** menu and clicking **Save Points to Workspace**. Save the points by overwriting the `cpstruct` variable.

```
n = 2;
% inputImageDownsample = inputImage(1:n:end,1:n:end,1);
% baseGrayDownsample = baseGray(1:n:end,1:n:end);
% cpselect(inputImageDownsample,baseGrayDownsample)
```

Apply Geometric Transformation to Aerial Photograph

Extract the control point pairs from the control point structure. Account for the downsampling.

```
[input,base] = cpstruct2pairs(cpstruct);
input = n*input - 1;
base = n*base - 1;
```

Convert the coordinates of the base image to map (State Plane) coordinates.

```
[x,y] = intrinsicToWorld(R,base(:,1),base(:,2));
spatial = [x y];
```

Create a projective geometric transformation that you can use to align the orthophoto image with the aerial image. A projective transformation is a reasonable choice because the aerial image is from a frame camera and the terrain for the region is fairly gentle.

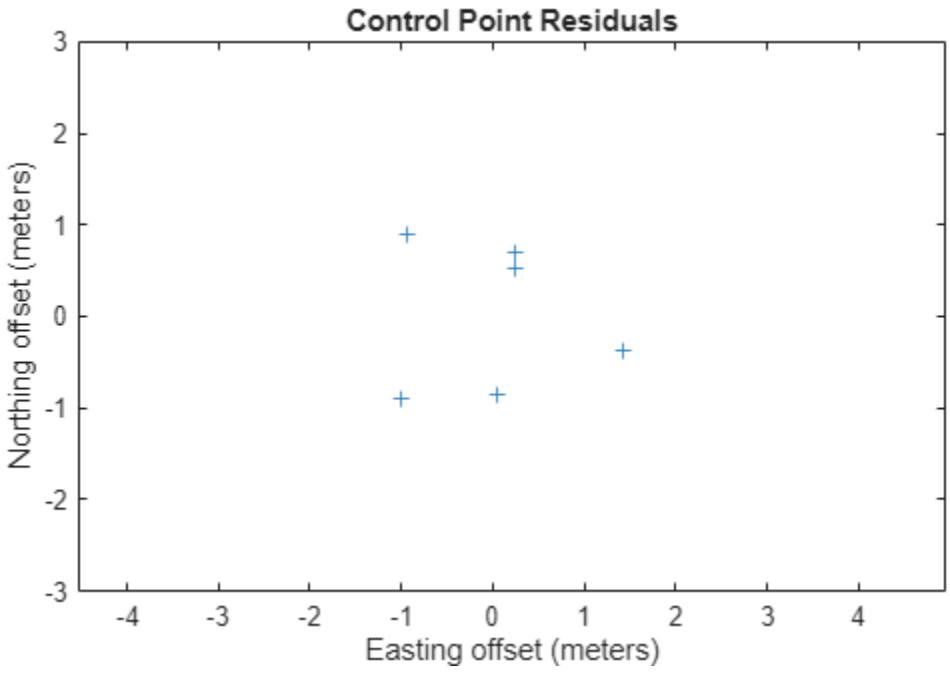
```
tform = fitgeotform2d(input,spatial,"projective");
```

Compute the 2-D residuals.

```
residuals = transformPointsForward(tform,input) - spatial;
```

Plot the 2-D residuals. Add a title and axes labels. Adjust the axes limits.

```
figure
plot(residuals(:,1),residuals(:,2),"+")

title("Control Point Residuals")
xlabel("Easting offset (meters)")
ylabel("Northing offset (meters)")

xlim([-3 3])
ylim([-3 3])
axis equal
```

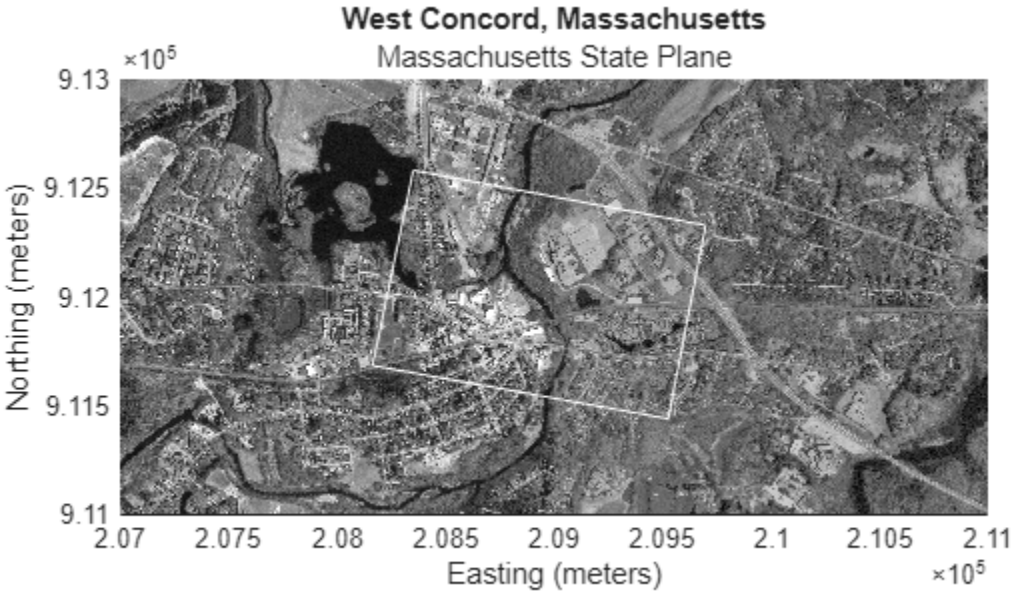


Predict the corner point locations of the registered image in map coordinates.

```
mInput = size(inputImage,1);
nInput = size(inputImage,2);

inputCorners = 0.5 + [0 0; 0 mInput; nInput mInput; nInput 0; 0 0];

outputCornersSpatial = transformPointsForward(tform,inputCorners);
outputCornersX = outputCornersSpatial(:,1);
outputCornersY = outputCornersSpatial(:,2);
```

Display the outline of the image on the map by connecting the corner points.

```
figure(ax1.Parent)
line(outputCornersX,outputCornersY,Color="w")
```



Calculate the average pixel size of the input image in map units.

```
pixelSize = [hypot( ...
    outputCornersX(2) - outputCornersX(1), ...
    outputCornersY(2) - outputCornersY(1)) / mInput, ...
 hypot( ...
    outputCornersX(4) - outputCornersX(5), ...
    outputCornersY(4) - outputCornersY(5)) / nInput]
```

```
pixelSize = 1×2

    0.9020    0.8959
```

The average pixel size provides a starting point from which to select a width and height for the pixels in the georeferenced output image. In principle, the output pixels can be any size. However, if you make the pixel size too small, you might waste memory and computation time, and create a blurry image. If you make the pixel size too big, then you might alias the image or needlessly discard

information from the original image. In general, you also want the pixels to be square. A reasonable heuristic is to select a "reasonable" number (i.e., 0.95 or 1.0 rather than 0.9020) that is slightly larger than the maximum average pixel size.

For this example, specify a pixel size of 1, which means that each pixel in the georeferenced image covers one square meter on the ground. Georeferenced images that align along integer map coordinates are convenient for display and analysis.

```
outputPixelSize = 1;
```

Choose world limits that are integer multiples of the output pixel size.

```
xWorldLimits = outputPixelSize ...
    * [floor(min(outputCornersX) / outputPixelSize), ...
       ceil(max(outputCornersX) / outputPixelSize)]
```

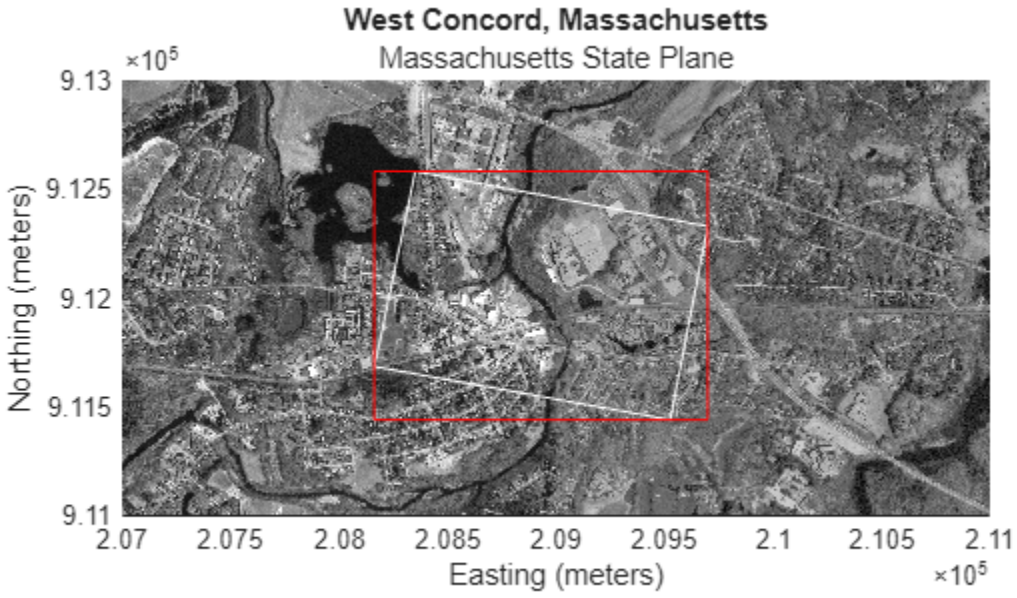
```
xWorldLimits = 1×2

      208163      209699
```

```
yWorldLimits = outputPixelSize ...
    * [floor(min(outputCornersY) / outputPixelSize), ...
       ceil(max(outputCornersY) / outputPixelSize)]
```

```
yWorldLimits = 1×2

      911437      912581
```

Display the bounding box for the registered image.

```
line(xWorldLimits([1 1 2 2 1]),yWorldLimits([2 1 1 2 2]),Color="red")
```



Calculate the dimensions of the registered image.

```
mOutput = diff(yWorldLimits) / outputPixelSize
```

```
mOutput =
1144
```

```
nOutput = diff(xWorldLimits) / outputPixelSize
```

```
nOutput =
1536
```

Create an Image Processing Toolbox referencing object for the registered image.

```
Rimage = imref2d([mOutput nOutput],xWorldLimits,yWorldLimits)
```

```
Rimage =
  imref2d with properties:

           XWorldLimits: [208163 209699]
           YWorldLimits: [911437 912581]
              ImageSize: [1144 1536]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 1536
    ImageExtentInWorldY: 1144
       XIntrinsicLimits: [0.5000 1.5365e+03]
       YIntrinsicLimits: [0.5000 1.1445e+03]
```

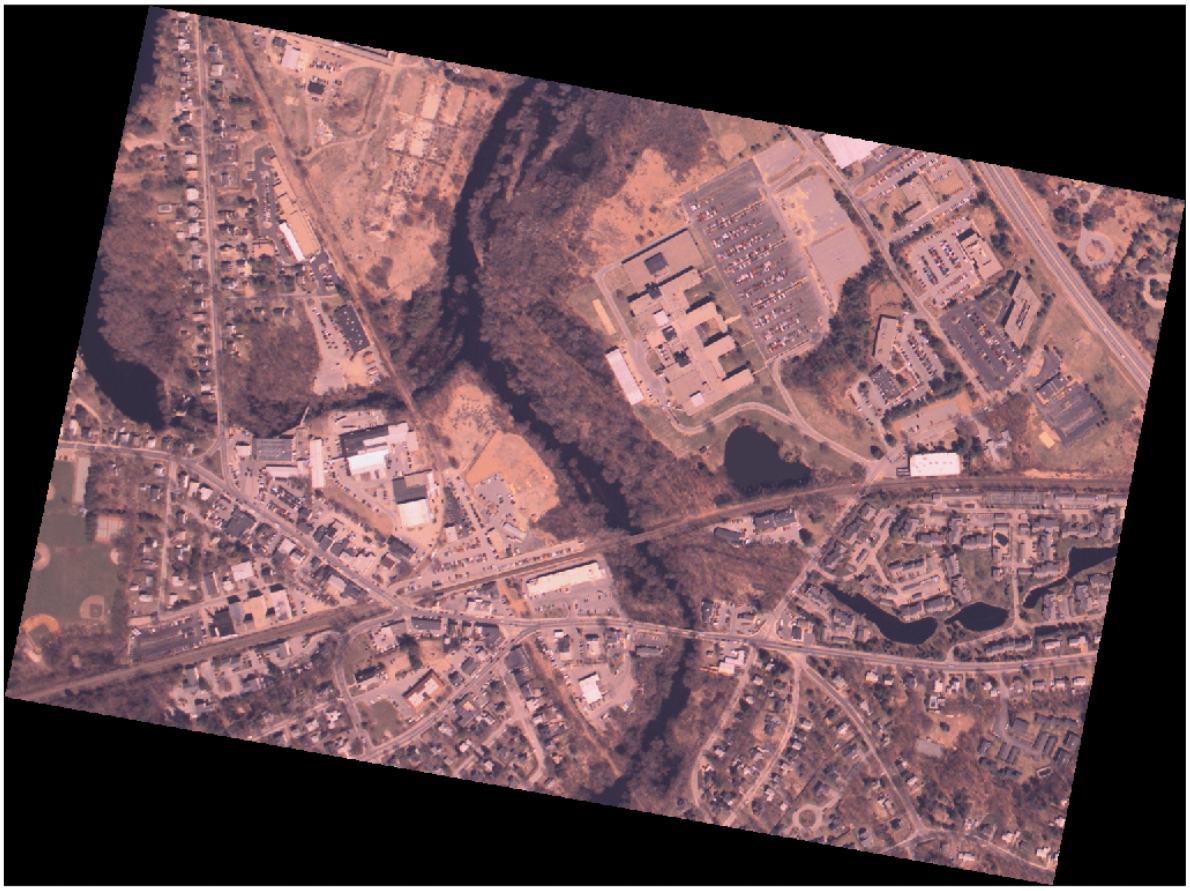
Create a map raster reference object for the registered image, which is the Mapping Toolbox counterpart to an Image Processing Toolbox referencing object.

```
Rmap = maprefcells(Rimage.XWorldLimits,Rimage.YWorldLimits,Rimage.ImageSize, ...
    ColumnsStartFrom="north")
```

```
Rmap =
  MapCellsReference with properties:

            XWorldLimits: [208163 209699]
            YWorldLimits: [911437 912581]
              RasterSize: [1144 1536]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1
      CellExtentInWorldY: 1
    RasterExtentInWorldX: 1536
    RasterExtentInWorldY: 1144
        XIntrinsicLimits: [0.5 1536.5]
        YIntrinsicLimits: [0.5 1144.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
            ProjectedCRS: []
```

Apply the geometric transformation by using the `imwarp` function. Flip the output so that the columns run from north to south.

```
registered = flipud(imwarp(inputImage,tform,OutputView=Rimage));

figure
imshow(registered)
```



Write the registered image to a TIFF file. Write the spatial referencing information to a world file.

```
imwrite(registered,"concord_aerial_sw_reg.tif");
worldfilewrite(Rmap,getworldfilename("concord_aerial_sw_reg.tif"));
```

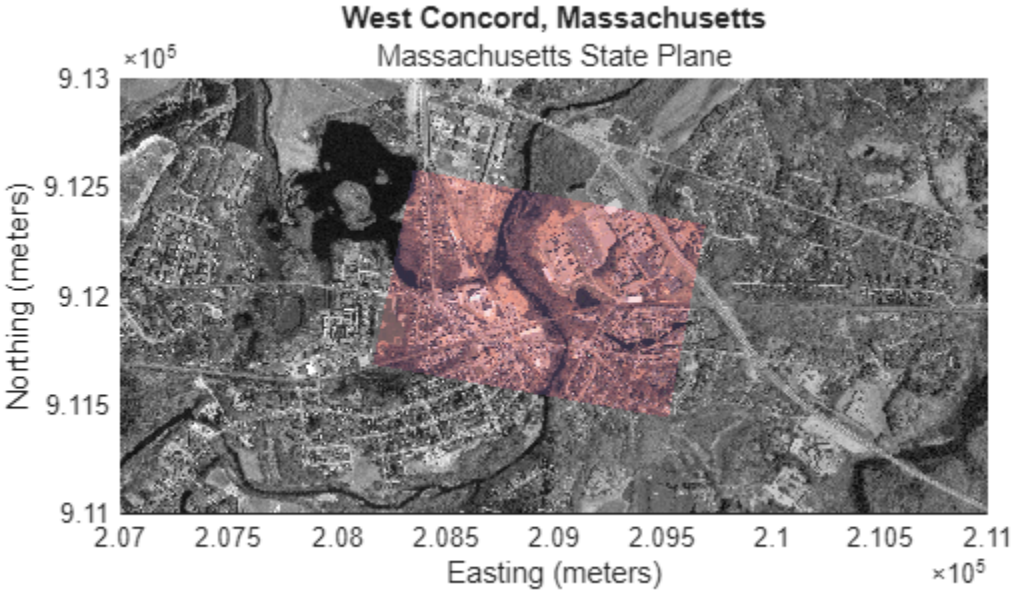
Display Registered Image in Map Coordinates

Create a new map of the orthophoto base tiles.

```
figure
mapshow(baseImage,cmap,R)

title("West Concord, Massachusetts")
subtitle("Massachusetts State Plane")

xlabel("Easting (meters)")
ylabel("Northing (meters)")
```

Display the registered image on the same map. The registered image does not completely fill its bounding box, so the image includes missing data. Render the missing data as transparent by creating an alpha data mask. You can assess the registration by looking at features that span both the registered image and the orthophoto images.

```
alphaData = registered(:,:,1);
alphaData(alphaData~=0) = 255;

ax2 = gca;
mInput = mapshow(ax2,registered,Rmap);
mInput.AlphaData = alphaData;
```



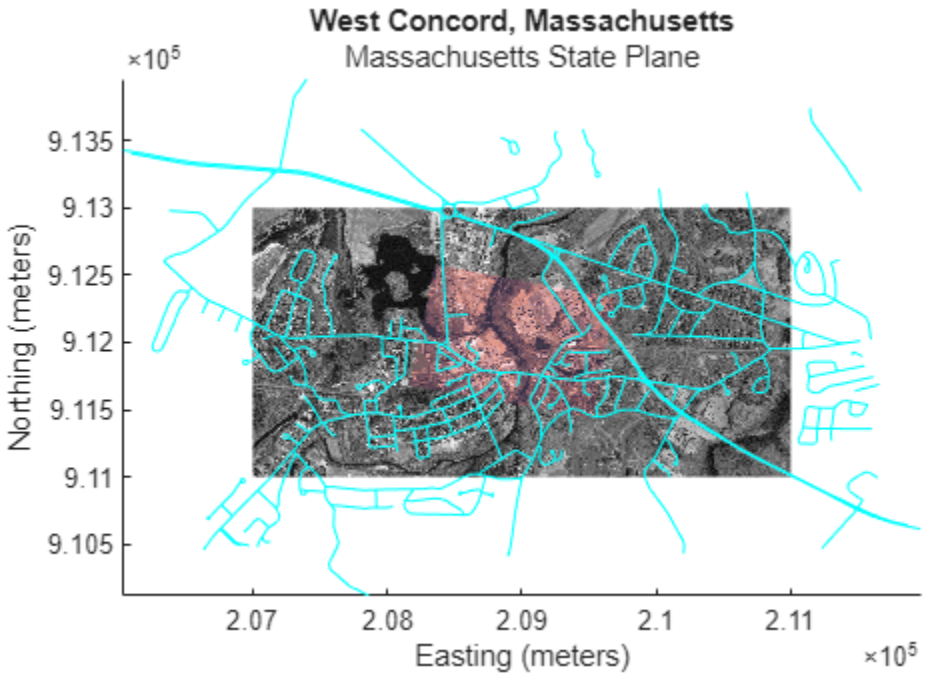
Overlay Vector Road Layer

Specify the name of a shapefile that contains road data for an area around West Concord. Get information about the shapefile by using the `shapeinfo` function. Read the data into the workspace as a geospatial table by using the `readgeotable` function.

```
roadsfile = "concord_roads.shp";
roadinfo = shapeinfo(roadsfile);
roads = readgeotable(roadsfile);
```

Create a symbol specification that hides minor roads. Then, display the roads on the same map. Note that the roads in the shapefile generally align with the roads in the images. The two obvious linear features that are not represented by the shapefile data are railroads.

```
roadspec = makesymbolspec("Line",{'CLASS',6,'Visible','off'});
mapshow(roads,SymbolSpec=roadspec,Color="cyan")
```



See Also

Functions
`readgeoraster` | `im2uint8` | `cpstruct2pairs` | `intrinsicToWorld` | `fitgeotform2d` | `transformPointsForward` | `imref2d`

Objects
`MapCellsReference`

Unproject a Digital Elevation Model (DEM)

This example shows how to convert a USGS DEM into a regular latitude-longitude grid having comparable spatial resolution. U.S. Geological Survey (USGS) 30-meter Digital Elevation Models (DEMs) are regular grids (raster data) that use the UTM coordinate system. Using such DEMs in applications may require reprojecting and resampling them. You can readily apply the approach show here to projected map coordinate systems other than UTM and to other DEMs and most types of regular data grids.

Import DEM

This example uses a USGS DEM for a quadrangle 7.5-arc-minutes square located in the White Mountains of New Hampshire, USA. Import the data and a map cells reference object using the `readgeoraster` function. Prepare the data for plotting by specifying the output type as `"double"`.

```
[Z,R] = readgeoraster("MtWashington-ft.grd",OutputType="double");
```

Get Projection Information

Get information about the projected coordinate reference system by querying the `ProjectedCRS` property of the reference object. The result is a `projcrs` object. Then, get the ellipsoid for the coordinate reference system.

```
p = R.ProjectedCRS;
ellipsoid = p.GeographicCRS.Spheroid
```

```
ellipsoid =
referenceEllipsoid with defining properties:

                 Code: 7008
                 Name: 'Clarke 1866'
           LengthUnit: 'meter'
        SemimajorAxis: 6378206.4
        SemiminorAxis: 6356583.8
    InverseFlattening: 294.978698213898
         Eccentricity: 0.0822718542230038

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Determine UTM Zone and Create Map

From the `Name` property of the `projcrs` object, you can tell that the DEM is gridded in a Universal Transverse Mercator (UTM) coordinate system.

```
p.Name
```

```
ans =
"UTM Zone 19, Northern Hemisphere"
```

To find the UTM zone, first locate the center of the DEM in UTM coordinates. Then, convert the coordinates to latitude-longitude.

```
[M,N] = size(Z);
xCenterIntrinsic = (1 + N)/2;
yCenterIntrinsic = (1 + M)/2;
[xCenter,yCenter] = intrinsicToWorld(R,xCenterIntrinsic,yCenterIntrinsic);
[latCenter,lonCenter] = projinv(p,xCenter,yCenter)
```

```
latCenter =
44.3124

lonCenter =
-71.3126
```

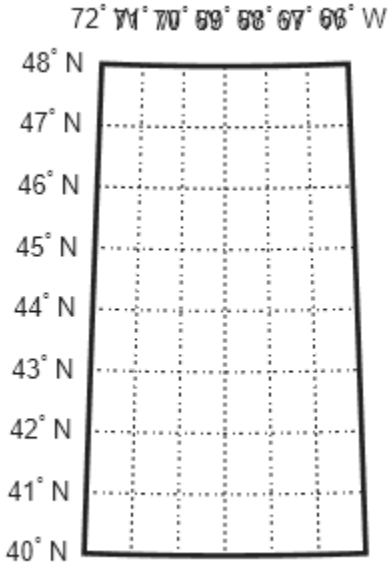
Find the UTM zone for the DEM by using the `utmzone` function.

```
utmZone = utmzone(latCenter,lonCenter)
```

```
utmZone =
'19T'
```

Use the zone and ellipsoid to create an `axesm`-based map.

```
figure
axesm("utm",Zone=utmZone,Geoid=ellipsoid)
axis off
gridm
mlabel on
plabel on
framem on
```

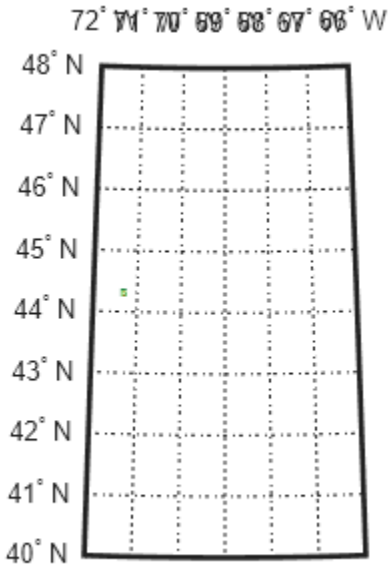


If you can visually place the approximate location of New Hampshire on a world map, then you can confirm this result by using the `utmzoneui` function: `utmzoneui(actualZone)`.

Display DEM on Map

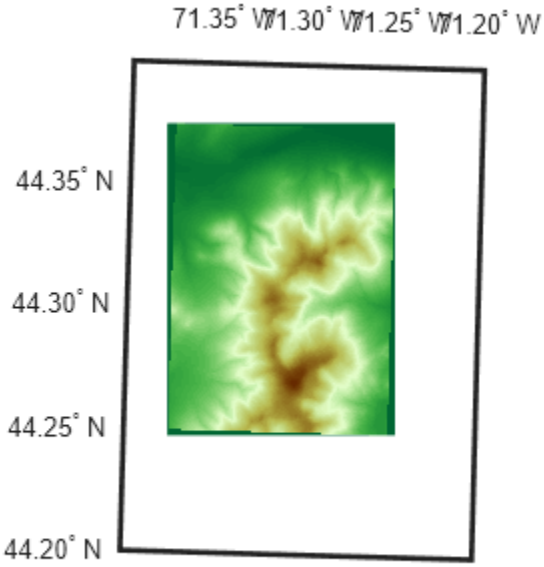
Use `mapshow` (rather than `geoshow` or `meshm`) to display the DEM on the map because the data are gridded in map (x-y) coordinates.

```
mapshow(Z,R,DisplayType="texturemap")
demcmap(Z)
```



The DEM covers such a small part of this map that it may be hard to see (look between 44 and 44 degrees North and 72 and 71 degrees West), because the map limits are set to cover the entire UTM zone. You can reset them (as well as the map grid and label parameters) to get a closer look.

```
setm(gca,"MapLatLimit",[44.2 44.4],"MapLonLimit",[-71.4 -71.2])
setm(gca,"MLabelLocation",0.05,"MLabelRound",-2)
setm(gca,"PLabelLocation",0.05,"PLabelRound",-2)
setm(gca,"PLineLocation",0.025,"MLineLocation",0.025)
```



When it is viewed at this larger scale, narrow wedge-shaped areas of uniform color appear along the edge of the grid. These are places where Z contains the value NaN, which indicates the absence of actual data. By default they receive the first color in the color table, which in this case is dark green. These null-data areas arise because although the DEM is gridded in UTM coordinates, its data limits are defined by a latitude-longitude quadrangle. The narrow angle of each wedge corresponds to the non-zero "grid declination" of the UTM coordinate system in this part of the zone. (Lines of constant x run precisely north-south only along the central meridian of the zone. Elsewhere, they follow a slight angle relative to the local meridians.)

Define Output Latitude-Longitude Grid

The next step is to define a regularly-spaced set of grid points in latitude-longitude that covers the area within the DEM at about the same spatial resolution as the original data set.

First, you need to determine how the latitude changes between rows in the input DEM (i.e., by moving northward by 30 meters).

```
rng = R.CellExtentInWorldY;  % In meters, consistent with p.LengthUnit
latcrad = deg2rad(latCenter);   % latCenter in radians

% Change in latitude, in degrees
dLat = rad2deg(meridianfwd(latcrad,rng,ellipsoid) - latcrad)
```

```
dLat =
2.6998e-04
```

The actual spacing can be rounded slightly to define the grid spacing to be used for the output (latitude-longitude) grid.

```
gridSpacing = 1/4000;   % In other words, 4000 samples per degree
```

To set the extent of the output (latitude-longitude) grid, start by finding the corners of the DEM in UTM map coordinates.

```
xCorners = R.XWorldLimits([1 1 2 2])'
```

```
xCorners = 4×1

      310380
      310380
      320730
      320730
```

```
yCorners = R.YWorldLimits([1 2 2 1])'
```

```
yCorners = 4×1

     4901880
     4916040
     4916040
     4901880
```

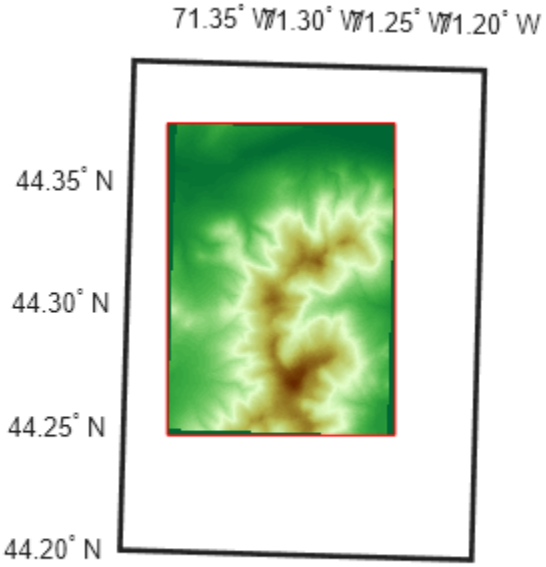
Then convert the corners to latitude-longitude. Display the latitude-longitude corners on the map (via the UTM projection) to check that the results are reasonable.

```
[latCorners, lonCorners] = projinv(p,xCorners, yCorners)
```

```
latCorners = 4×1

   44.2474
   44.3748
   44.3774
   44.2500
```

```
lonCorners = 4×1

  -71.3749
  -71.3801
  -71.2502
  -71.2454
```

```
hCorners = geoshow(latCorners,lonCorners,DisplayType="polygon",...
    FaceColor="none",EdgeColor="red");
```



Next, round outward to define an output latitude-longitude quadrangle that fully encloses the DEM and aligns with multiples of the grid spacing.

```
latSouth = gridSpacing * floor(min(latCorners)/gridSpacing)
```

```
latSouth =
44.2473
```

```
lonWest  = gridSpacing * floor(min(lonCorners)/gridSpacing)
```

```
lonWest =
-71.3803
```

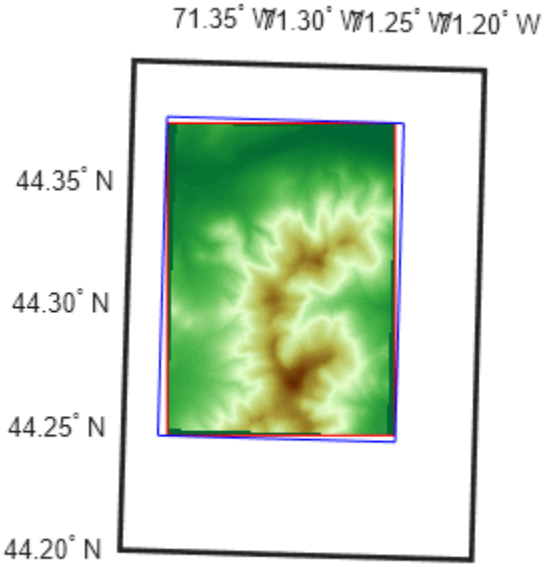
```
latNorth = gridSpacing * ceil( max(latCorners)/gridSpacing)
```

```
latNorth =
44.3775
```

```
lonEast  = gridSpacing * ceil( max(lonCorners)/gridSpacing)
```

```
lonEast =
-71.2452
```

```
qlatlim = [latSouth latNorth];
qlonlim = [lonWest lonEast];

dlat = 100*gridSpacing;
dlon = 100*gridSpacing;

[latquad, lonquad] = outlinegeoquad(qlatlim, qlonlim, dlat, dlon);

hquad = geoshow(latquad, lonquad, ...
    DisplayType="polygon",FaceColor="none",EdgeColor="blue");
```



Finally, construct a geographic raster referencing object for the output grid. It supports transformations between latitude-longitude and the row and column subscripts. In this case, use of a world file matrix, W, enables exact specification of the grid spacing. Display W with more decimal places by temporarily changing the default display format.

```
currentFormat = format; % since R2021a
format longG
W = [gridSpacing    0              lonWest + gridSpacing/2; ...
     0              gridSpacing    latSouth + gridSpacing/2]
```

```
W = 2×3

                   0.00025                         0                -71.380125
                         0                 0.00025                 44.247375
```

```
format(currentFormat)
nRows = round((latNorth - latSouth) / gridSpacing)
```

```
nRows =
521
```

```
nCols = round(wrapTo360(lonEast - lonWest) / gridSpacing)
```

```
nCols =
540
```

```
Rlatlon = georasterref(W,[nRows nCols],"cells");
Rlatlon.GeographicCRS = p.GeographicCRS
```

```
Rlatlon =
  GeographicCellsReference with properties:

             LatitudeLimits: [44.24725 44.3775]
```

```
          LongitudeLimits: [-71.38025 -71.24525]
               RasterSize: [521 540]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'south'
           RowsStartFrom: 'west'
    CellExtentInLatitude: 1/4000
   CellExtentInLongitude: 1/4000
  RasterExtentInLatitude: 0.13025
 RasterExtentInLongitude: 0.135
        XIntrinsicLimits: [0.5 540.5]
        YIntrinsicLimits: [0.5 521.5]
    CoordinateSystemType: 'geographic'
           GeographicCRS: [1×1 geocrs]
               AngleUnit: 'degree'
```

`Rlatlon` fully defines the number and location of each cell in the output grid.

Map Each Output Grid Point Location to UTM X-Y

Finally, you're ready to make use of the map projection, applying it to the location of each point in the output grid. First compute the latitudes and longitudes of those points, stored in 2-D arrays.

```
[rows,cols] = ndgrid(1:nRows, 1:nCols);
[lat,lon] = intrinsicToGeographic(Rlatlon,cols,rows);
```

Then apply the projection to each latitude-longitude pair, arrays of UTM x-y locations having the same shape and size as the latitude-longitude arrays.

```
[XI,YI] = projfwd(p,lat,lon);
```

At this point, `XI(i,j)` and `YI(i,j)` specify the UTM coordinate of the grid point corresponding to the i-th row and j-th column of the output grid.

Resample Original DEM

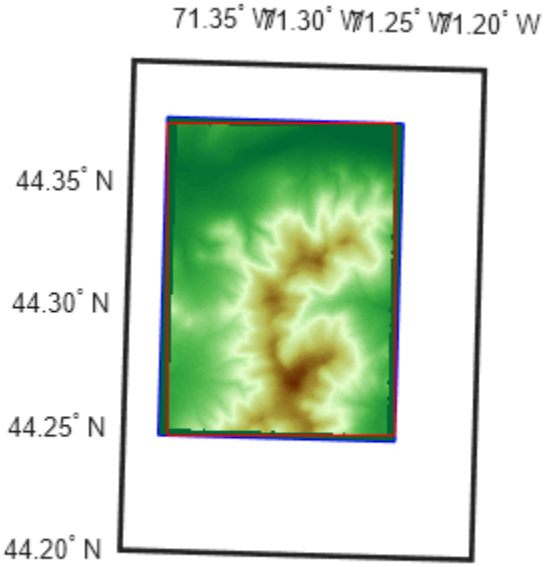
The final step is to use the MATLAB `interp2` function to perform bilinear resampling.

At this stage, the value of projecting from the latitude-longitude grid into the UTM map coordinate system becomes evident: it means that the resampling can take place in the regular X-Y grid, making `interp2` applicable. The reverse approach, unprojecting each (X,Y) point into latitude-longitude, might seem more intuitive but it would result in an irregular array of points to be interpolated -- a much harder task, requiring use of the far more costly `griddata` function or some rough equivalent.

```
[rows,cols] = ndgrid(1:M,1:N);
[X,Y] = intrinsicToWorld(R,cols,rows);
method = "linear";
extrapval = NaN;
Zlatlon = interp2(X,Y,Z,XI,YI,method,extrapval);
```

View the result in the projected axes using `geoshow`, which will re-project it on the fly. Notice that it fills the blue rectangle, which is aligned with lines of latitude and longitude. (In contrast, the red rectangle, which outlines the original DEM, aligns with UTM x and y.) Also notice NaN-filled regions along the edges of the grid. The boundaries of these regions appear slightly jagged, at the level of a single grid spacing, due to round-off effects during interpolation. Move the red quadrilateral and blue quadrangle to the top, to ensure that they are not hidden by the raster display.

```
geoshow(Zlatlon,Rlatlon,DisplayType="texturemap")
uistack([hCorners hquad],"top")
```



See Also

`refmatToMapRasterReference` | `intrinsicToWorld` | `intrinsicToGeographic` | `georasterref` | `demcmap`

Define Areas of Interest

An area of interest (AOI) is a type of region of interest (ROI) that defines a focus area for geospatial analysis and visualization. For example, you can use AOIs to indicate the boundaries of a city, to represent the footprint of an elevation data file, or to define a wireless coverage zone.

Mapping Toolbox™ represents AOIs using polygon shape objects.

- Represent an AOI with coordinates in a geographic coordinate reference system (CRS) by using a scalar `geopolyshape` object.
- Represent an AOI with coordinates in a projected CRS by using a scalar `mappolyshape` object.

Because AOIs are `geopolyshape` and `mappolyshape` objects, you can examine and modify AOIs by using the corresponding object functions. For example, you can find the area of an AOI by using the `area` function, buffer an AOI by using the `buffer` function, and find the bounds of an AOI by using the `bounds` function.

You can define AOIs in several ways. You can:

- Define a quadrangular or rectangular AOI by using the `aoiquad` function.
- Define a circular AOI by using the `aoicircle` function.
- Define an AOI from polygon shapes by using the `union` function.
- Define an AOI from vertices by using the `geopolyshape` or `mappolyshape` function.
- Define an AOI from the name of a region, state, county, or physical feature by using the `geocode` function.

These examples show how to define AOIs using each of the listed processes.

Quadrangular or Rectangular AOI

Define a quadrangular or rectangular AOI by using the `aoiquad` function. The function defines the AOI by finding the bounds of the specified data. The AOI is a quadrangle when the data is in geographic coordinates and a rectangle when the data is in projected coordinates.

Get information about a raster data file by creating a `RasterInfo` object. Extract the spatial reference from the object. The spatial reference indicates that the raster is a grid of rectangular cells referenced to projected coordinates.

```
info = georasterinfo("MountDartmouth-ft.grd");
R = info.RasterReference
```

```
R =
  MapCellsReference with properties:

           XWorldLimits: [300390 310770]
           YWorldLimits: [4902150 4916340]
             RasterSize: [473 346]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 30
      CellExtentInWorldY: 30
    RasterExtentInWorldX: 10380
```

```
    RasterExtentInWorldY: 14190
        XIntrinsicLimits: [0.5 346.5]
        YIntrinsicLimits: [0.5 473.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
            ProjectedCRS: [1×1 projcrs]
```

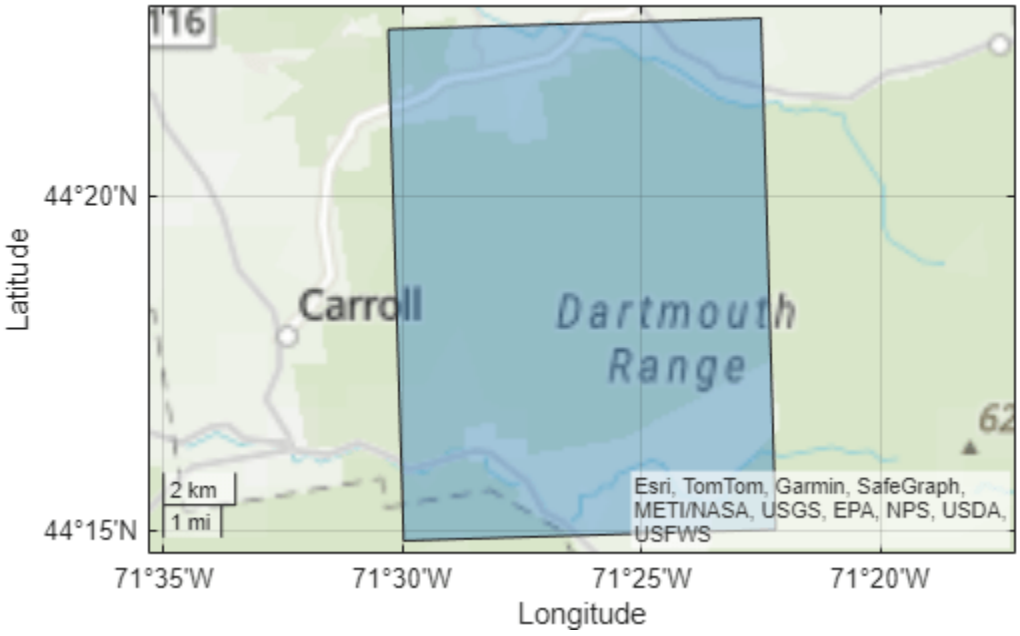
Create a rectangular AOI from the spatial reference. When you pass data in projected coordinates to the `aoiquad` function, the function represents the AOI using a scalar polygon shape in projected coordinates.

```
aoiR = aoiquad(R)
```

```
aoiR =
  mappolyshape with properties:

                NumRegions: 1
                  NumHoles: 0
                  Geometry: "polygon"
    CoordinateSystemType: "planar"
            ProjectedCRS: [1×1 projcrs]
```

Display the AOI on a map. The AOI represents the footprint of the raster data file.

```
figure
geobasemap topographic
geoplot(aoiR)
```



The `aoiquad` function also enables you to create AOIs from vector data. For more examples, see the `aoiquad` reference page.

Circular AOI

Define a circular AOI by using the `aoicircle` function. The function defines the AOI by creating circles with a specified radius that are centered at the specified coordinates.

Read a shapefile containing the coordinates of locations in Boston into a geospatial table. The table represents the locations using point shapes in projected coordinates.

```
places = readgeotable("boston_placenames.shp");
shape = places.Shape
```

```
shape =
  13×1 mappointshape array with properties:

               NumPoints: [13×1 double]
                       X: [13×1 double]
                       Y: [13×1 double]
                Geometry: "point"
    CoordinateSystemType: "planar"
            ProjectedCRS: [1×1 projcrs]
```

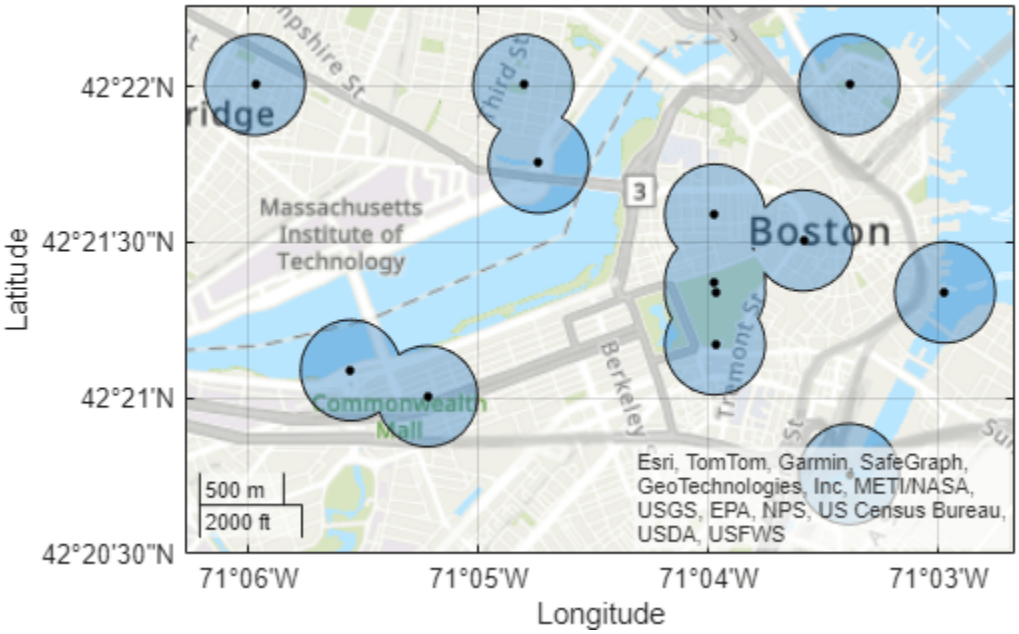
Create an AOI of circles from the point shapes. Specify the radius of each circle as 300 meters. When you pass data in projected coordinates to the `aoicircle` function, the function represents the AOI using a scalar polygon shape in projected coordinates.

```
aoiC = aoicircle(shape,300)
```

```
aoiC =
  mappolyshape with properties:

              NumRegions: 7
                NumHoles: 1
                Geometry: "polygon"
    CoordinateSystemType: "planar"
            ProjectedCRS: [1×1 projcrs]
```

Display the AOI and the locations on a map.

```
figure
geoplot(aoiC)
hold on
geobasemap topographic
geoplot(shape,"k",MarkerSize=10)
```



The `aoicircle` function also enables you to create AOIs from numeric coordinates. For more examples, see the `aoicircle` reference page.

AOI from Polygon Shapes

Define an AOI from polygon shapes by using the `union` function. The function defines the AOI by combining the polygon shapes into a scalar polygon shape.

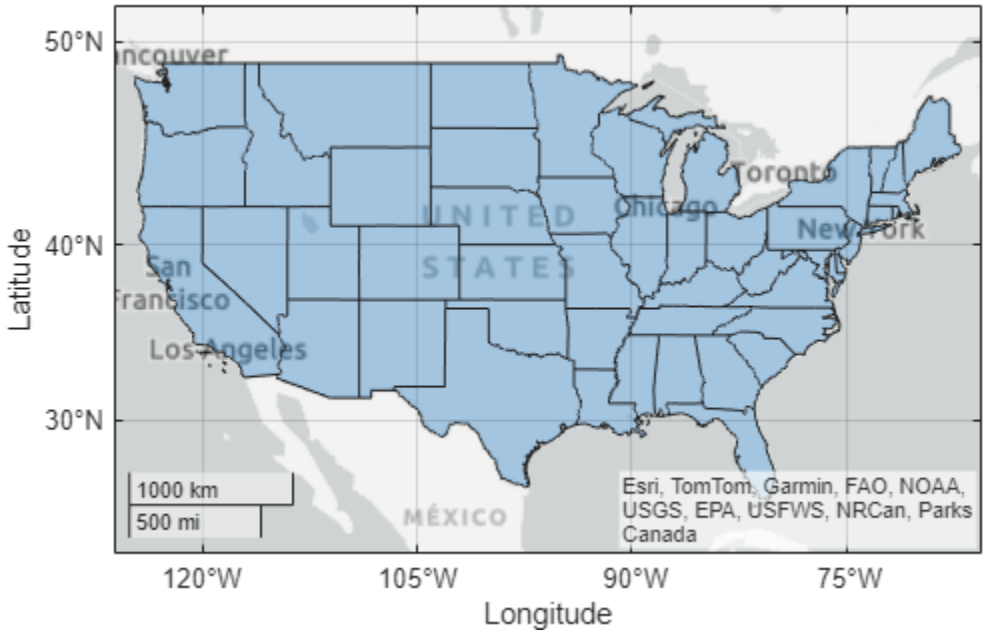
Read a shapefile of US states into the workspace as a geospatial table. The table represents the states using polygon shapes in geographic coordinates. Create a subtable that contains the states in the conterminous US. Then, extract the polygon shapes.

```
states = readgeotable("usastatehi.shp");
idx = states.Name ~= "Hawaii" & states.Name ~= "Alaska";
conus = states(idx,:);

pshapes = conus.Shape
```

```
pshapes=49×1 geopolyshape array with properties:
              NumRegions: [49×1 double]
                NumHoles: [49×1 double]
                Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
      ⋮
```

Display the polygon shapes on a map.

```
figure
geoplot(pshapes)
```



Create an AOI from the polygon shapes. When you pass an array of polygon shapes to the `union` function, the function creates a scalar polygon shape.

```
aoiP = union(pshapes)
```

```
aoiP =
  geopolyshape with properties:

             NumRegions: 51
               NumHoles: 0
               Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Display the AOI on a new map.

```
figure
geoplot(aoiP)
```

AOI from Vertices

Define an AOI from vertices by using the `geopolyshape` or `mappolyshape` function. These functions define the AOI by creating polygons with vertices at the specified coordinates.

Create an AOI from vertices in geographic coordinates by using the `geopolyshape` function.

View satellite imagery for a region surrounding the MathWorks Apple Hill campus in Natick, MA.

```
figure
geobasemap satellite
geolimits([42.2971 42.3024],[-71.3562 -71.3453])
```

Identify the vertices of a seven-sided polygon that surrounds the campus. To use predefined vertices, specify `interactivelySelectVertices` as `false`. To interactively select the vertices, specify `interactivelySelectVertices` as `true`.

```
interactivelySelectVertices = false;
if interactivelySelectVertices
    [lat,lon] = ginput(7);
else
    lat = [42.3005 42.2993 42.2992 42.2987 42.2984 42.2986 42.3012];
    lon = [-71.3537 -71.3536 -71.3528 -71.3528 -71.3511 -71.3482 -71.3481];
end
```

Close the polygon by appending the first vertex to the end of the list of vertices.

```
lat(end+1) = lat(1);
lon(end+1) = lon(1);
```

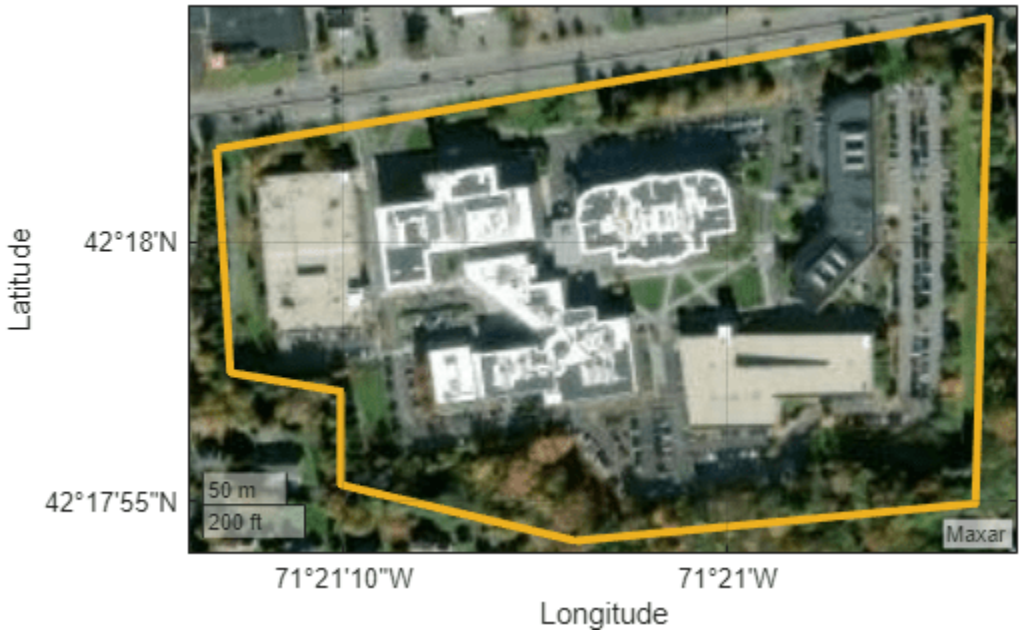
Create an AOI from the vertices. Specify the geographic CRS as the World Geodetic System of 1984, which has the EPSG code 4326.

```
aoiV = geopolyshape(lat,lon);
aoiV.GeographicCRS = geocrs(4326)
```

```
aoiV =
  geopolyshape with properties:

             NumRegions: 0
               NumHoles: 1
               Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Display the boundary of the AOI on the map.

```
geoplot(aoiV,FaceColor="none",LineWidth=3,EdgeColor="#EDB120")
```



Create an AOI from vertices in projected coordinates by using the `mappolyshape` function. For more examples, see the `mappolyshape` reference page.

AOI from Placename

Define an AOI from the placename for a region, state, county, or physical feature by using the `geocode` function. The function gets the AOI by converting the name into a polygon shape.

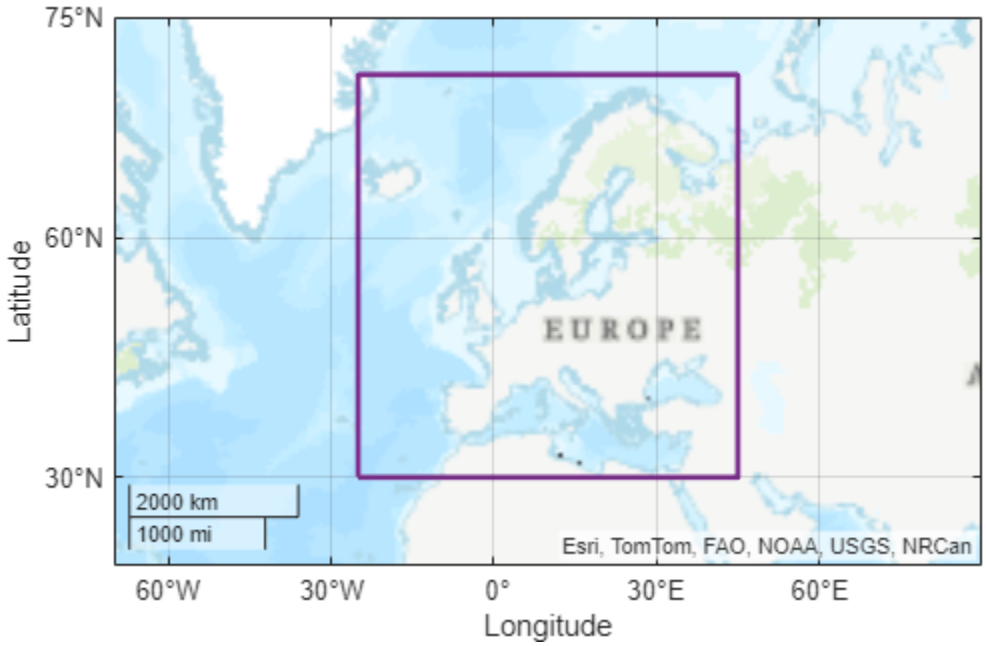
Get an AOI for the world region of Europe. The `geocode` function creates a geospatial table that contains the AOI and the name of the matched placename. Extract the AOI from the table.

```
europeGT = geocode("Europe");
aoiE = europeGT.Shape
```

```
aoiE =
  geopolyshape with properties:

             NumRegions: 1
               NumHoles: 0
               Geometry: "polygon"
    CoordinateSystemType: "geographic"
           GeographicCRS: []
```

When you geocode the name of a world region, the AOI represents either an outline of the region or a quadrangle near the region. Display the boundary of the AOI on a map. Provide more geographic context for the AOI by zooming out.

```
figure
geobasemap topographic
geoplot(aoiE,LineWidth=2,EdgeColor="#7E2F8E",FaceColor="none")
geolimits([15 75],[-70 90])
```



The `geocode` function also enables you to get AOIs from custom tables. For more examples, see the `geocode` reference page.

See Also

Functions
`geoplot` | `readgeotable` | `georasterinfo` | `ginput`

Objects
`geopolyshape` | `mappolyshape`

Related Examples

- “Clip Vector Data to Area of Interest” on page 6-91
- “Crop Raster Data to Area of Interest” on page 6-95
- “Query Points in Area of Interest” on page 6-102

Clip Vector Data to Area of Interest

This topic shows how to clip vector data to an area of interest (AOI). Within the topic, you:

- Read vector data from a file into a geospatial table. The geospatial table represents the data using point, line, and polygon shape objects.
- Define an AOI. Mapping Toolbox™ represents AOIs using polygon shape objects. For information about creating AOIs, see “Define Areas of Interest” on page 6-82.
- Clip the vector data to the AOI by using the `geoclip` or `mapclip` function.

To clip vector data to an AOI, these properties of the vector shape objects and the AOI must match:

- The type of coordinate system. You can get the type of coordinate system for a shape object by querying the `CoordinateSystemType` property.
- The coordinate reference system (CRS). For shape objects in a geographic coordinate system, you can get the CRS by querying the `GeographicCRS` property. For shape objects in a planar coordinate system, you can get the CRS by querying the `ProjectedCRS` property.

These examples show how to clip geographic and projected shape objects to an AOI.

Clip Geographic Shapes to AOI

Clip vector data in geographic coordinates to an AOI by using the `geoclip` function.

Read vector data from a shapefile of world cities into a geospatial table. The table represents the cities using point shapes in geographic coordinates. Extract the point shapes.

```
cities = readgeotable("worldcities.shp");
cShape = cities.Shape
```

```
cShape =
  318×1 geopointshape array with properties:

               NumPoints: [318×1 double]
                Latitude: [318×1 double]
               Longitude: [318×1 double]
                Geometry: "point"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Define a circular AOI with a radius of 11 degrees that is centered on Denver, Colorado.

```
denver = geocode("Denver, CO","city");
dAOI = aoicircle(denver,11);
```

Verify that the city data and the AOI use the same type of coordinate system.

```
isequal(cShape.CoordinateSystemType,dAOI.CoordinateSystemType)
```

```
ans = logical
   1
```

Clip the city data to the AOI.

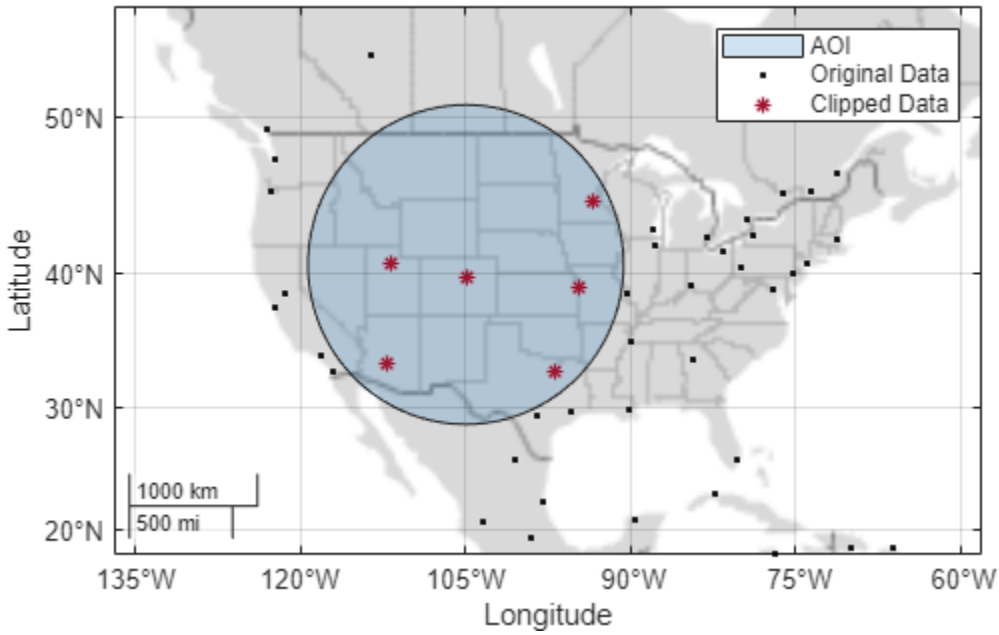
```
gClippedG = geoclip(cShape,dAOI);
```

Display the AOI, the original city data, and the clipped city data on a map. Prepare to add a legend by specifying a display name for each plot.

```
figure
geobasemap grayland
geoplot(dAOI,FaceAlpha=0.2,DisplayName="AOI")
hold on
geoplot(cShape,"k",DisplayName="Original Data")
geoplot(gClippedG,"*",MarkerEdgeColor="#A2142F",DisplayName="Clipped Data")
```

Add a legend. Get a clearer view of the clipped data by zooming in.

```
legend
geolimits([18 56],[-137 -58])
```



Clip Projected Shapes to AOI

Clip vector data in projected coordinates to an AOI by using the `mapclip` function.

Read vector data from a shapefile of Boston roads into a geospatial table. The table represents the roads using line shapes in projected coordinates. Extract the line shapes.

```
roads = readgeotable("boston_roads.shp");
rShape = roads.Shape
```

```
rShape=2795×1 maplineshape array with properties:
                NumParts: [2795×1 double]
                Geometry: "line"
    CoordinateSystemType: "planar"
            ProjectedCRS: [1×1 projcrs]
      ⋮
```

Read a shapefile containing the coordinates of locations in Boston as a geospatial table. The table represents the locations using point shapes in projected coordinates. Create a subtable by geocoding four of the locations. Then, define a quadrangular AOI that bounds the locations.

```
locations = readgeotable("boston_placenames.shp");
sublocations = geocode(["BOSTON","BOSTON NECK","BEACON HILL","COPPS HILL"],locations);

bAOI = aoiquad(sublocations)
```

```
bAOI =
  mappolyshape with properties:

             NumRegions: 1
               NumHoles: 0
               Geometry: "polygon"
    CoordinateSystemType: "planar"
           ProjectedCRS: [1×1 projcrs]
```

Verify that the road data and the AOI use the same type of coordinate system.

```
isequal(rShape.CoordinateSystemType,bAOI.CoordinateSystemType)
```

```
ans = logical
   1
```

Verify that the road data and the AOI use the same projected CRS.

```
isequal(rShape.ProjectedCRS,bAOI.ProjectedCRS)
```

```
ans = logical
   1
```

Clip the road data to the AOI.

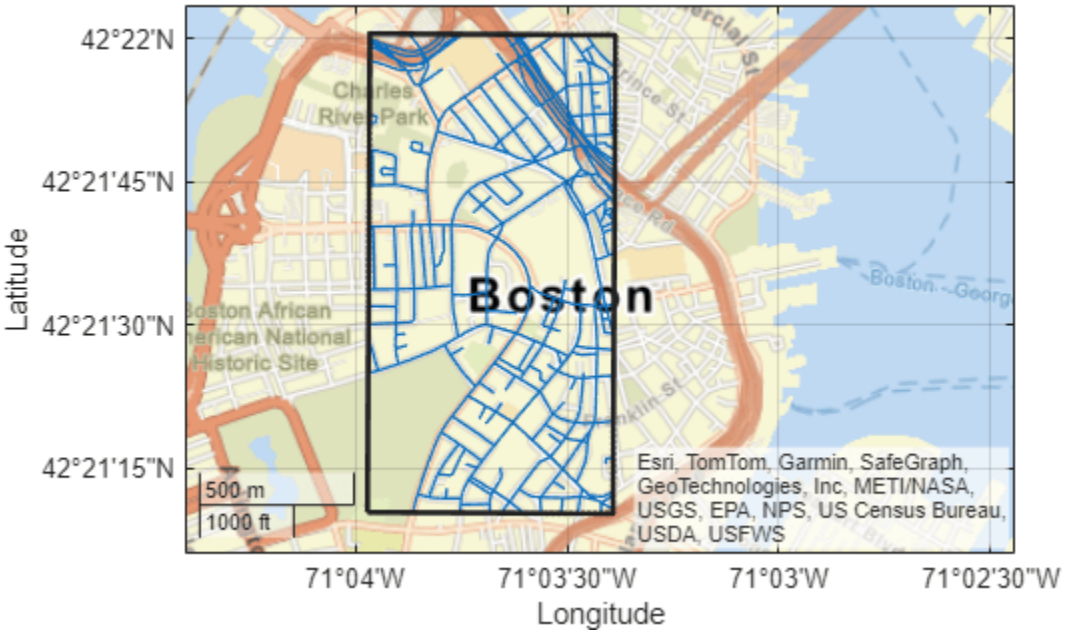
```
rClipped = mapclip(rShape,bAOI);
```

Display the clipped road data and the boundary of the AOI on a map.

```
figure
geobasemap streets
geoplot(rClipped,LineWidth=1)
hold on
geoplot(bAOI,FaceColor="none",LineWidth=2)
```

Provide more geographic context for the data by zooming out.

```
geolimits([42.3517 42.3676],[-71.0734 -71.0407])
```



See Also

Functions
`aoiquad` | `aoicircle` | `geocode` | `geoplot` | `readgeotable`

Objects
`geopolyshape` | `mappolyshape`

Related Examples

- "Define Areas of Interest" on page 6-82
- "Crop Raster Data to Area of Interest" on page 6-95
- "Create Geospatial Tables" on page 6-2

Crop Raster Data to Area of Interest

This topic shows how to crop raster data to an area of interest (AOI). Within the topic, you:

- Read raster data from a file into an array and a spatial reference object.
- Define a quadrangular or rectangular AOI by using the `aoiquad` and `buffer` functions. Mapping Toolbox™ represents AOIs using polygon shape objects.
- Get the bounds of the AOI by using the `bounds` function.
- Crop the raster data to the bounds of the AOI by using the `geocrop` or `mapcrop` function.

To crop raster data to an AOI, these properties of the raster data and the AOI must match:

- The type of coordinate system. You can get the type of coordinate system for a reference object or a shape object by querying the `CoordinateSystemType` property.
- The coordinate reference system (CRS). For reference objects and shape objects in a geographic coordinate system, you can get the CRS by querying the `GeographicCRS` property. For objects in a planar coordinate system, you can get the CRS by querying the `ProjectedCRS` property.

These examples show how to crop geographic and projected raster data to an AOI.

Crop Geographic Raster to AOI

Crop raster data in geographic coordinates to an AOI by using the `geocrop` function.

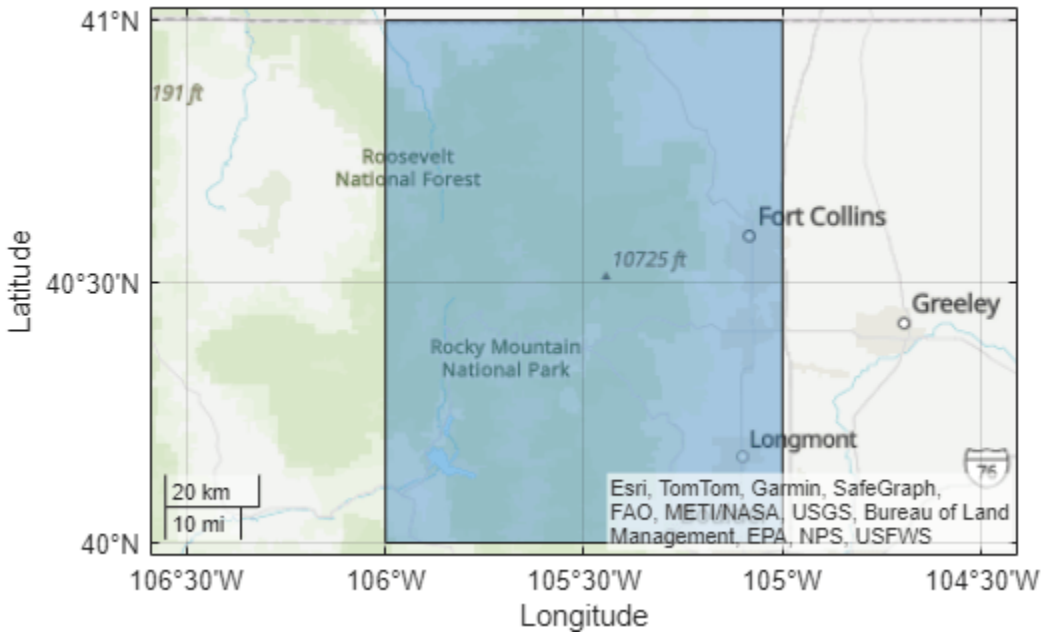
Read elevation data from a DTED file into an array and a raster reference object. The reference object indicates that the raster is referenced to a grid of posting points in geographic coordinates.

```
[dtedZ,dtedR] = readgeoraster("n40_w106_3arc_v2.dt1",OutputType="double");
disp(dtedR)
```

```
  GeographicPostingsReference with properties:

            LatitudeLimits: [40 41]
           LongitudeLimits: [-106 -105]
                RasterSize: [1201 1201]
       RasterInterpretation: 'postings'
           ColumnsStartFrom: 'north'
             RowsStartFrom: 'west'
    SampleSpacingInLatitude: 1/1200
   SampleSpacingInLongitude: 1/1200
     RasterExtentInLatitude: 1
    RasterExtentInLongitude: 1
           XIntrinsicLimits: [1 1201]
           YIntrinsicLimits: [1 1201]
      CoordinateSystemType: 'geographic'
             GeographicCRS: [1×1 geocrs]
                 AngleUnit: 'degree'
```

View the footprint of the DTED file on a map.

```
dtedFP = aoiquad(dtedR);
figure
geobasemap topographic
geoplot(dtedFP)
```



Identify the locations of three points within the footprint. To use predefined points, specify `interactivelySelectPoints` as `false`. To interactively select the points, specify `interactivelySelectPoints` as `true`.

```
interactivelySelectPoints = false;
if interactivelySelectPoints
    [lat,lon] = ginput(3);
else
    lat = [40.5130 40.4274 40.6064];
    lon = [-105.4439 -105.2516 -105.2427];
end
```

Create a quadrangular AOI from the points. Specify the geographic CRS as the World Geodetic System of 1984, which has the EPSG code `4326`.

```
gcrs = geocrs(4326);
pointsAOI = aoiquad(lat,lon,gcrs);
```

Ensure the points are within the AOI, instead of on the boundary of the AOI, by adding a buffer of 0.05 degrees around the AOI.

```
pointsAOI = buffer(pointsAOI,0.05);
```

Verify that the DTED data and the AOI use the same type of coordinate system.

```
isequal(dtedR.CoordinateSystemType,pointsAOI.CoordinateSystemType)
```

```
ans = logical
   1
```

Verify that the DTED data and the AOI use the same geographic CRS.

```
isequal(dtedR.GeographicCRS,pointsAOI.GeographicCRS)
```

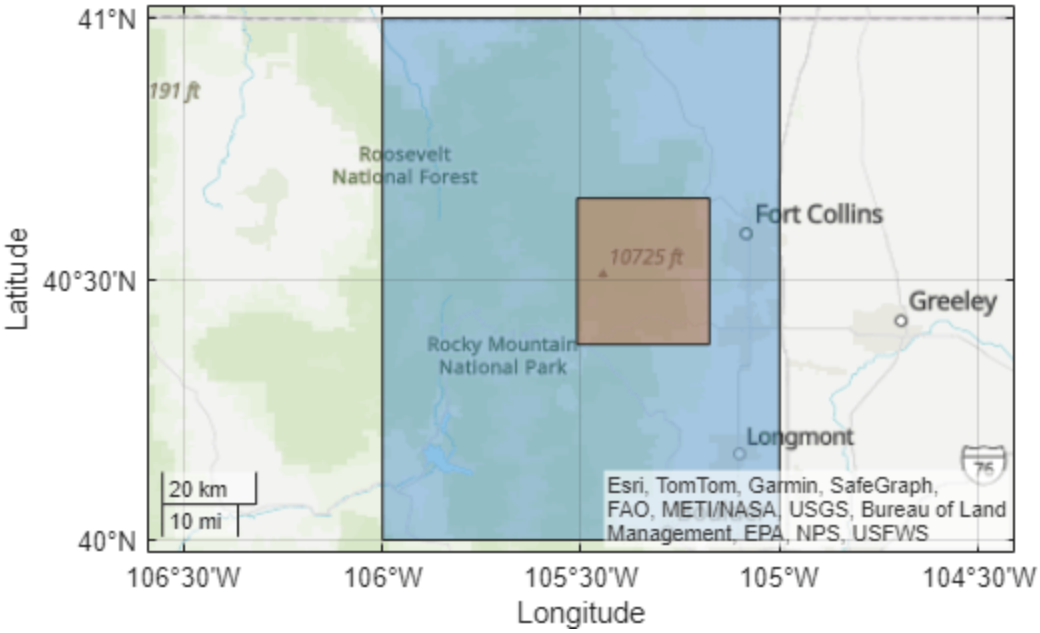
```
ans = logical
   1
```

Get the bounds of the AOI. Then, crop the DTED data to the bounds of the AOI.

```
[latlim,lonlim] = bounds(pointsAOI);
[dtedZcrop,dtedRcrop] = geocrop(dtedZ,dtedR,latlim,lonlim);
```

Display the footprint of the cropped DTED data over the footprint of the original DTED data.

```
dtedFPcrop = aoiquad(dtedRcrop);
hold on
geoplot(dtedFPcrop)
```

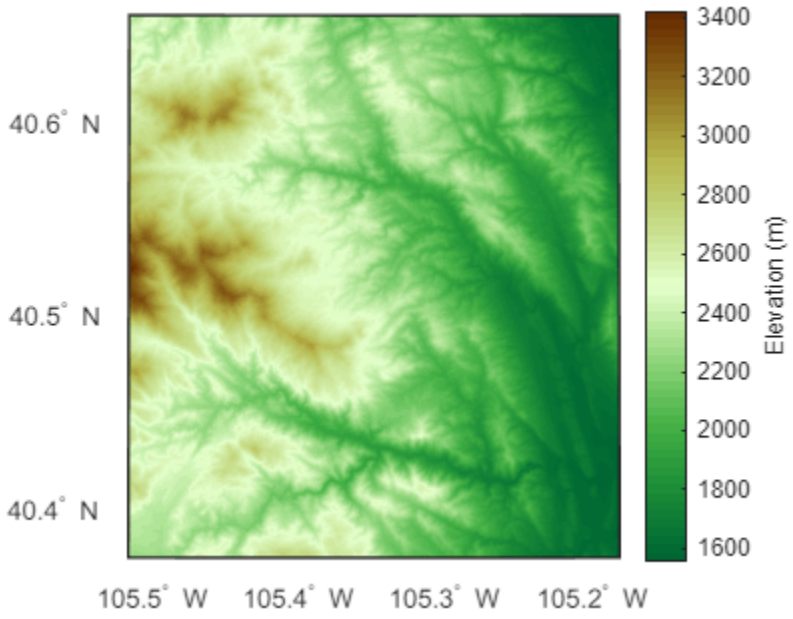


Using a new map, display the cropped DTED data as a surface. Unlike `geoplot`, the `usamap` and `geoshow` functions enable you to display geographic raster data on a map. For more information about selecting a map display, see “Choose a 2-D Map Display” on page 3-2.

```
figure
usamap(dtedZcrop,dtedRcrop)
geoshow(dtedZcrop,dtedRcrop,DisplayType="surface")
```

Apply a colormap that is appropriate for elevation data. Add a labeled color bar.

```
demcmap(dtedZcrop)
c = colorbar;
c.Label.String = "Elevation (m)";
```



Crop Projected Data to AOI

Crop raster data in projected coordinates to an AOI by using the `mapcrop` function.

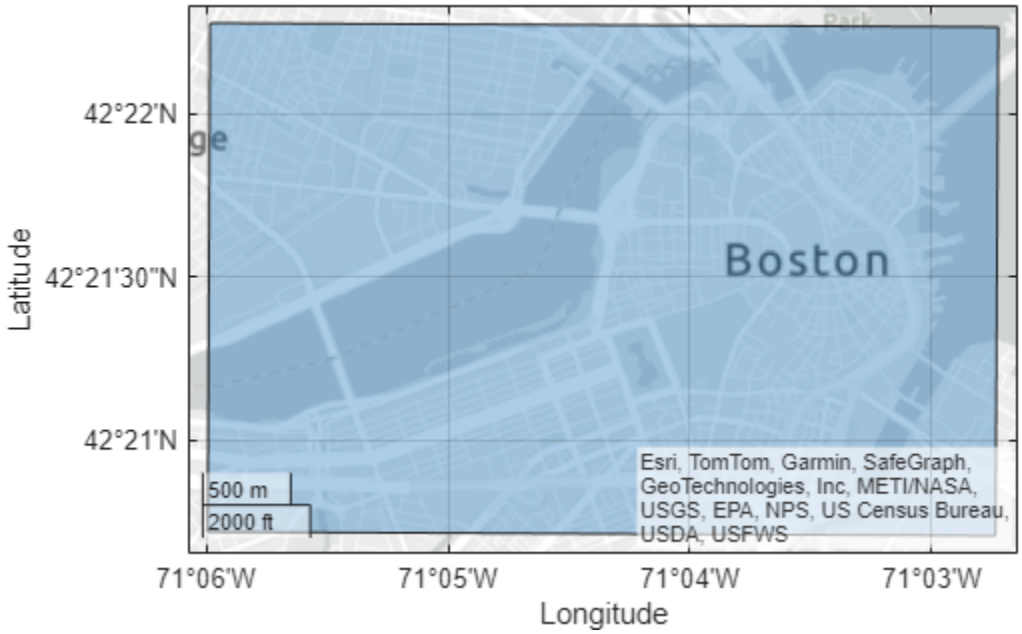
Read imagery of Boston into an array and a raster reference object. The reference object indicates that the raster is referenced to a grid of cells in projected coordinates.

```
[bostonA,bostonR] = readgeoraster("boston.tif");
disp(bostonR)
```

```
  MapCellsReference with properties:

          XWorldLimits: [764317.995579396 779019.409746063]
          YWorldLimits: [2951122.28480912 2960574.36564245]
            RasterSize: [2881 4481]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 3.28083333333333
      CellExtentInWorldY: 3.28083333333333
    RasterExtentInWorldX: 14701.4141666667
    RasterExtentInWorldY: 9452.08083333332
        XIntrinsicLimits: [0.5 4481.5]
        YIntrinsicLimits: [0.5 2881.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
           ProjectedCRS: [1×1 projcrs]
```

View the footprint of the imagery on a map.

```
bostonFP = aoiquad(bostonR);
figure
geoplot(bostonFP)
```



Read an outline of two Boston parks into a geospatial table. The table represents the parks using a polygon shape in projected coordinates.

```
parks = readgeotable("bostonparks.shp");
parks.Shape
```

```
ans =
  mappolyshape with properties:

              NumRegions: 1
                NumHoles: 0
                Geometry: "polygon"
    CoordinateSystemType: "planar"
           ProjectedCRS: [1×1 projcrs]
```

Create a rectangular AOI from the polygon shape. Add a buffer of 300 meters around the AOI.

```
parksAOI = aoiquad(parks);
parksAOI = buffer(parksAOI,300);
```

Verify that the imagery and the AOI use the same type of coordinate system.

```
isequal(bostonR.CoordinateSystemType,parksAOI.CoordinateSystemType)
```

```
ans = logical
   1
```

Verify that the imagery and the AOI use the same projected CRS.

```
isequal(bostonR.ProjectedCRS,parksAOI.ProjectedCRS)
```

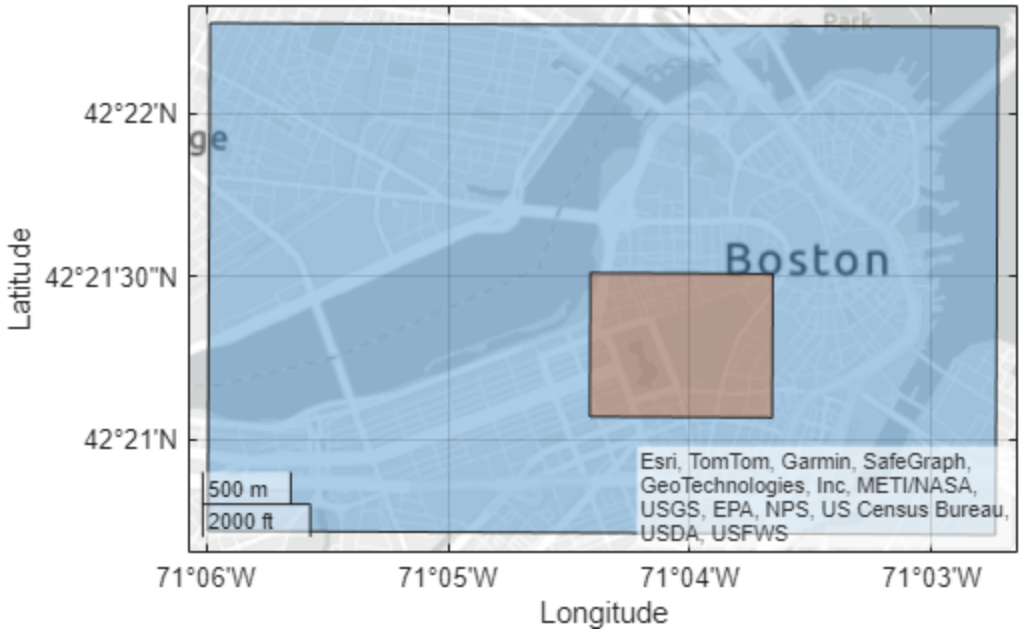
```
ans = logical
   1
```

Get the bounds of the AOI. Then, crop the imagery to the AOI.

```
[xlimits,ylimits] = bounds(parksAOI);
[bostonAcrop,bostonRcrop] = mapcrop(bostonA,bostonR,xlimits,ylimits);
```

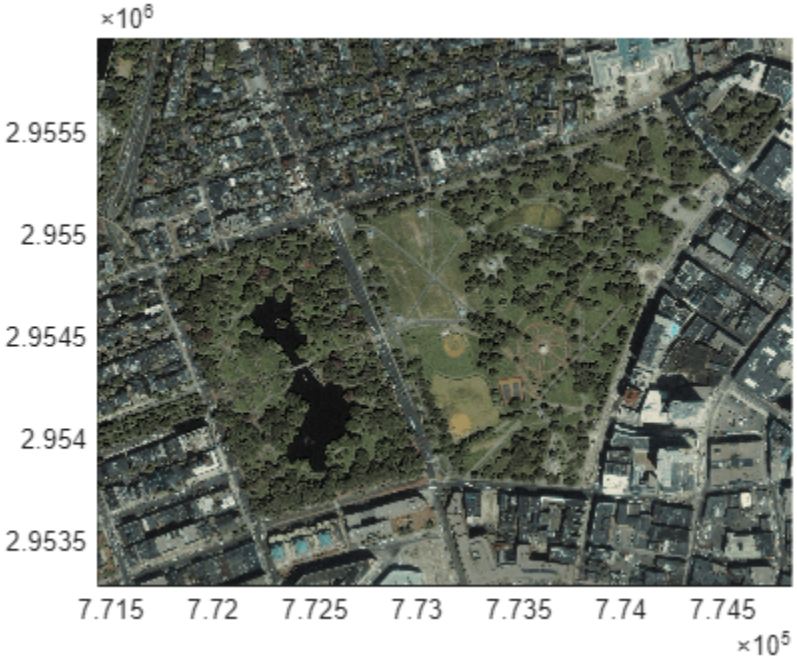
Display the footprint of the cropped imagery over the footprint of the original imagery.

```
bostonFPcrop = aoiquad(bostonRcrop);
hold on
geoplot(bostonFPcrop)
```



Using a new map, display the cropped imagery. Unlike `geoplot`, the `mapshow` function enables you to display projected raster data on a map. For more information about selecting a map display, see “Choose a 2-D Map Display” on page 3-2.

```
figure
mapshow(bostonAcrop,bostonRcrop)
```



See Also

Functions
`aoiquad` | `aoicircle` | `readgeoraster` | `readgeotable` | `ginput`

Objects
`geopolyshape` | `mappolyshape`

Related Examples

- “Clip Vector Data to Area of Interest” on page 6-91
- “Spatially Reference Imported Rasters” on page 6-41

Query Points in Area of Interest

This topic shows how to determine if points are within an area of interest (AOI). Within the topic, you:

- Specify the points to query using point shape objects.
- Define the AOI. Mapping Toolbox™ represents AOIs using polygon shape objects. For information about creating AOIs, see “Define Areas of Interest” on page 6-82.
- Determine whether the locations are within the AOI by using the `isinterior` function.

To query points within an AOI, these properties of the point shapes and the AOI must match:

- The type of coordinate system. You can get the type of coordinate system for a shape object by querying the `CoordinateSystemType` property.
- The coordinate reference system (CRS). For shape objects in a geographic coordinate system, you can get the CRS by querying the `GeographicCRS` property. For shape objects in a planar coordinate system, you can get the CRS by querying the `ProjectedCRS` property.

These examples show how to determine whether geographic and projected point objects are within an AOI.

Query Geographic Shapes in AOI

Geocode the names of several European cities. The result is a geospatial table that represents the cities using point shapes in geographic coordinates. Extract the point shapes.

```
names = ["Madrid","Dublin","London","Berlin","Rome", ...
    "Prague","Budapest","Milan","Amsterdam","Munich"];
cities = geocode(names,"city");
cShape = cities.Shape
```

```
cShape =
  10×1 geopointshape array with properties:

               NumPoints: [10×1 double]
                Latitude: [10×1 double]
               Longitude: [10×1 double]
                Geometry: "point"
    CoordinateSystemType: "geographic"
           GeographicCRS: [1×1 geocrs]
```

Geocode the city of Paris, France. Define a circular AOI with a radius of 6.5 degrees that is centered on Paris.

```
paris = geocode("Paris, France","city");
parisAOI = aoicircle(paris,6.5)
```

```
parisAOI =
  geopolyshape with properties:

             NumRegions: 1
               NumHoles: 0
               Geometry: "polygon"
    CoordinateSystemType: "geographic"
```

```
           GeographicCRS: []
```

Verify that the cities and the AOI use the same type of coordinate system.

```
isequal(cShape.CoordinateSystemType,parisAOI.CoordinateSystemType)
```

```
ans = logical
   1
```

Verify that the cities and the AOI use the same geographic CRS.

```
isequal(cShape.GeographicCRS,parisAOI.GeographicCRS)
```

```
ans = logical
   0
```

Determine if the cities are within the AOI. The `isinterior` function returns an array of logical values, where `1` indicates that the corresponding city is within the AOI, and `0` indicates that the corresponding city is not within the AOI.

```
inParisAOI = isinterior(parisAOI,cShape)
```

```
inParisAOI = 10×1 logical array

   0
   0
   1
   0
   0
   0
   0
   1
   1
   1
```

Create two arrays. The first array contains the cities that are in the AOI. The second array contains the cities that are not in the AOI.

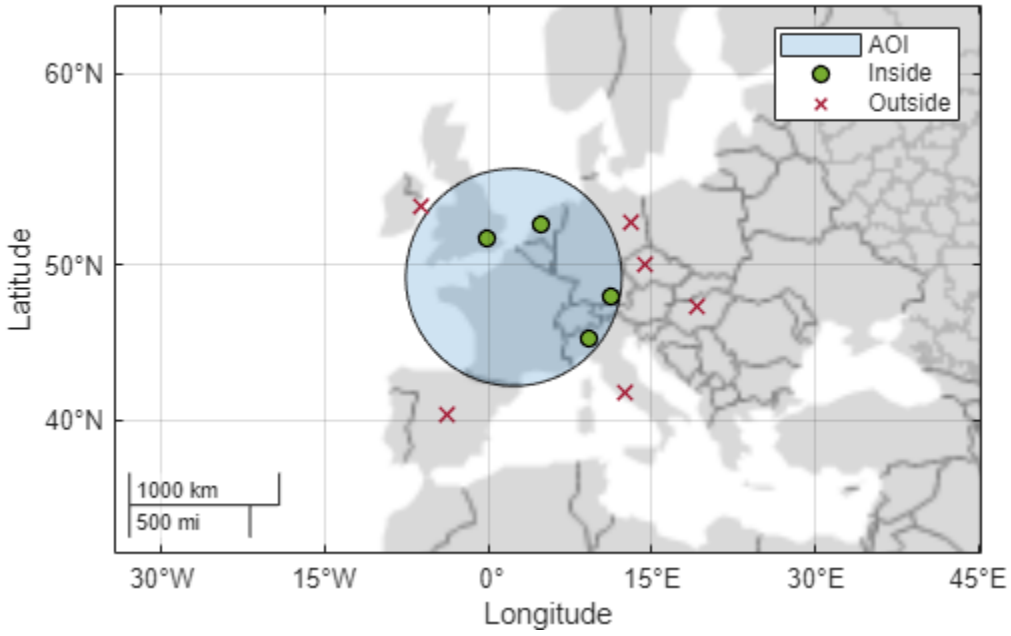
```
citiesInAOI = cShape(inParisAOI);
citiesOutAOI = cShape(~inParisAOI);
```

Display the AOI, the cities that are in the AOI, and the cities that are not in the AOI on a map. Prepare to add a legend by specifying a display name for each plot.

```
figure
geobasemap grayland
geoplot(parisAOI,FaceAlpha=0.2,DisplayName="AOI")
hold on
geoplot(citiesInAOI,"o",MarkerFaceColor="#77AC30",MarkerEdgeColor="k", ...
    DisplayName="Inside")
geoplot(citiesOutAOI,"x",MarkerEdgeColor="#A2142F",MarkerSize=8, ...
    DisplayName="Outside")
```

Add a legend. Provide more geographic context for the data by zooming out.

```
legend
geolimits([30 63],[-34 45])
```



Query Projected Shapes in AOI

Read a shapefile containing the coordinates of locations in Boston as a geospatial table. The table represents the locations using point shapes in projected coordinates. Extract the point shapes.

```
locations = readgeotable("boston_placenames.shp");
shape = locations.Shape
```

```
shape =
  13×1 mappointshape array with properties:

              NumPoints: [13×1 double]
                      X: [13×1 double]
                      Y: [13×1 double]
               Geometry: "point"
    CoordinateSystemType: "planar"
           ProjectedCRS: [1×1 projcrs]
```

Create a subtable by geocoding two of the locations. Then, create an AOI of circles from the locations. Specify the radius of each circle as 700 meters.

```
sublocations = geocode(["BEACON HILL","BROAD CANAL"],locations);
aoi = aoicircle(sublocations,700)
```

```
aoi =
  mappolyshape with properties:

             NumRegions: 1
               NumHoles: 0
               Geometry: "polygon"
```

```
    CoordinateSystemType: "planar"
            ProjectedCRS: [1×1 projcrs]
```

Verify that the locations and the AOI use the same type of coordinate system.

```
isequal(shape.CoordinateSystemType,aoi.CoordinateSystemType)
```

```
ans = logical
   1
```

Verify that the locations and the AOI use the same projected CRS.

```
isequal(shape.ProjectedCRS,aoi.ProjectedCRS)
```

```
ans = logical
   1
```

Determine if the locations are within the AOI. The `isinterior` function returns an array of logical values, where `1` indicates that the corresponding location is within the AOI, and `0` indicates that the corresponding location is not within the AOI.

```
inAOI = isinterior(aoi,shape)
```

```
inAOI = 13×1 logical array

   0
   0
   1
   1
   0
   1
   0
   0
   1
   1
   0
   1
   0
      ⋮
```

Create two arrays. The first array contains the locations that are in the AOI. The second array contains the locations that are not in the AOI.

```
locInAOI = shape(inAOI);
locOutAOI = shape(~inAOI);
```

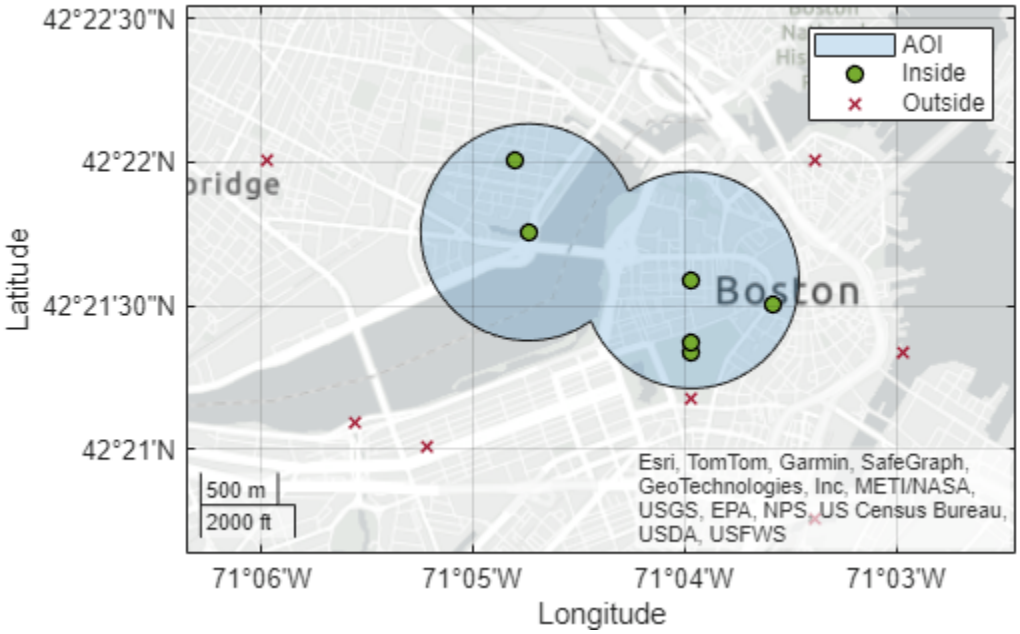
Display the AOI, the locations that are in the AOI, and the locations that are not in the AOI on a map. Prepare to add a legend by specifying a display name for each plot.

```
figure
geoplot(aoi,FaceAlpha=0.2,DisplayName="AOI")
hold on
geoplot(locInAOI,"o",MarkerFaceColor="#77AC30",MarkerEdgeColor="k", ...
    DisplayName="Inside")
```

```
geoplot(locOutAOI,"x",MarkerEdgeColor="#A2142F", ...
    DisplayName="Outside")
```

Add a legend and change the geographic limits.

```
legend
geolimits([42.3439 42.3758],[-71.1059 -71.0406])
```



See Also

Functions
`aoicircle` | `geoplot` | `readgeotable` | `geocode`

Objects
`geopolyshape` | `mappolyshape` | `geopointshape` | `mappointshape`

Convert Vector Data to Raster Format

You can convert latitude-longitude vector data to a grid at any resolution you choose to make a raster base map or grid layer. Certain Mapping Toolbox GUI tools help you do some of this, but you can also perform vector-to-raster conversions from the command line. The principal function for gridding vector data is `vec2mtx`, which allocates lines to a grid of any size you indicate, marking the lines with 1s and the unoccupied grid cells with 0s. The grid contains doubles, but if you want a logical grid (see “Data Grids as Logical Variables” on page 6-59) cast the result to be a logical array. To see an example, view “Creating Data Grids from Vector Data” on page 6-107.

If the vector data consists of polygons (patches), the gridded outlines are all hollow. You can differentiate them using the `encodem` function, calling it with an array of rows, columns, and seed values to produce a new grid containing polygonal areas filled with the seed values to replace the binary values generated by `vec2mtx`.

Creating Data Grids from Vector Data

This example shows how to convert vector data to raster data using the `vec2mtx` function. The example uses patch data for Indiana from the `usastatehi` shapefile. For more information, see “Convert Vector Data to Raster Format” on page 6-107.

Use `shaperead` to get the patch data for the boundary.

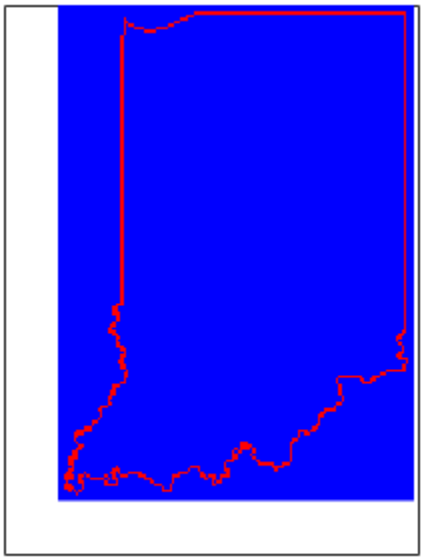
```
indiana = shaperead('usastatehi.shp',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi('Indiana',name),'Name'});
inLat = indiana.Lat;
inLon = indiana.Lon;
```

Convert the vectors to a regular data grid using `vec2mtx`. Set the grid density to be 40 cells per degree. Rasterize the boundary and generate a geographic raster reference object for it.

```
gridDensity = 40;
[inGrid,R] = vec2mtx(inLat,inLon,gridDensity);
```

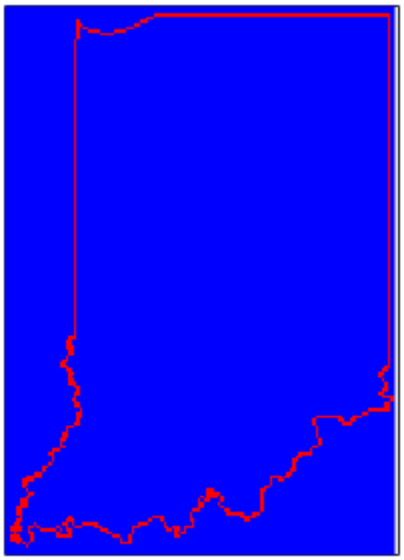
Make a map of the data grid in contrasting colors.

```
figure
axesm eqdcyl
meshm(inGrid,R)
colormap jet(4)
```



Set up the map limits.

```
setm(gca,'Flatlimit',R.LatitudeLimits,'FlonLimit',R.LongitudeLimits)
tightmap
```



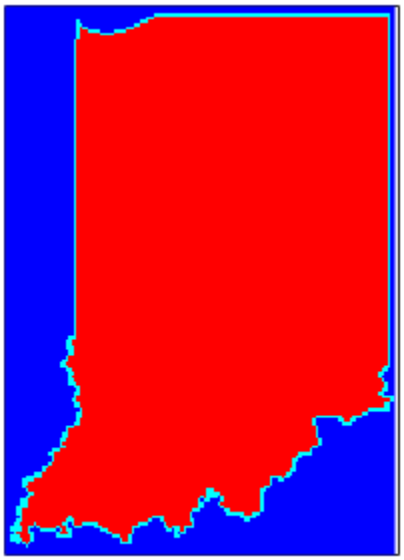
Specify the seed point and seed value. To fill (recode) the interior of Indiana, you need a seed point (which must be identified by row and column) and a seed value (to be allocated to all cells within the polygon). Select the middle row and column of the grid and choose an index value of 3 to identify the

territory when calling `encodem` to generate a new grid. The last argument (1) identifies the code for boundary cells, where filling should halt.

```
inPt = round([size(inGrid)/2,3]);
inGrid3 = encodem(inGrid,inPt,1);
```

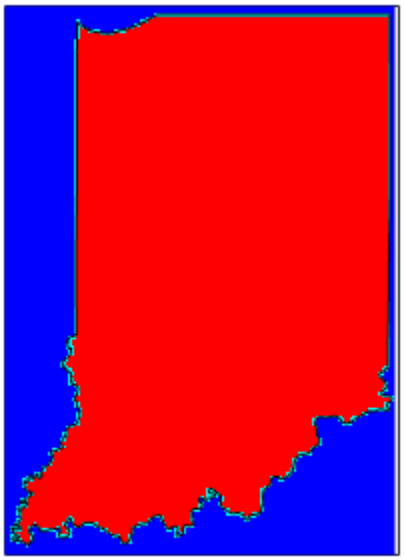
Clear and redraw the map using the filled grid.

```
meshm(inGrid3,R)
```



Plot the original vectors on the grid to see how well data was rasterized. You can use the Zoom tool on the figure window to examine the gridding results more closely.

```
plotm(inLat,inLon,'k')
```



Combine Vector and Raster Geodata on the Same Map

Vector map variables and data grid variables are often used or displayed together. For example, continental coastlines in vector form might be displayed with a grid of temperature data to make the latter more useful. When several map variables are used together, regardless of type, they can be referred to as a single map. To do this, of course, the different data sets must use the same coordinate system (i.e., geographic coordinates on the same ellipsoid or an identical map projection).

Combining Raster Data and Vector Data on the Same Map

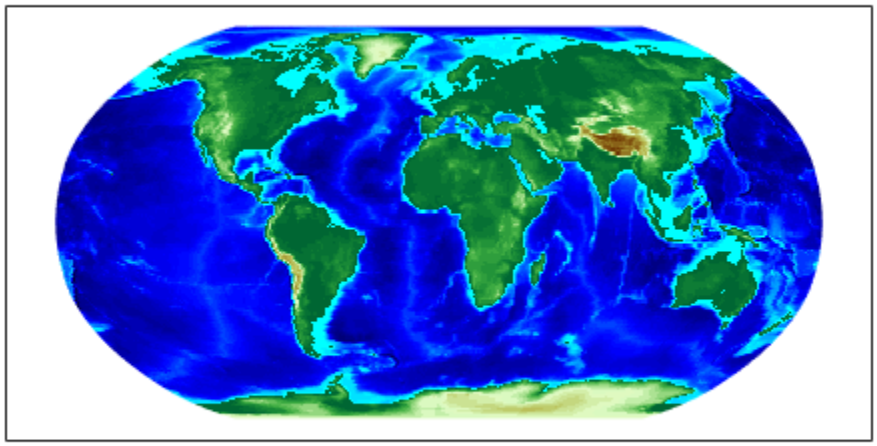
This example shows how to combine raster data and vector data on the same `axesm`-based map using the `geoshow` function.

First, load elevation raster data and a geographic cells reference object. Get the coordinates of coastlines as vectors.

```
load topo60c
load coastlines
```

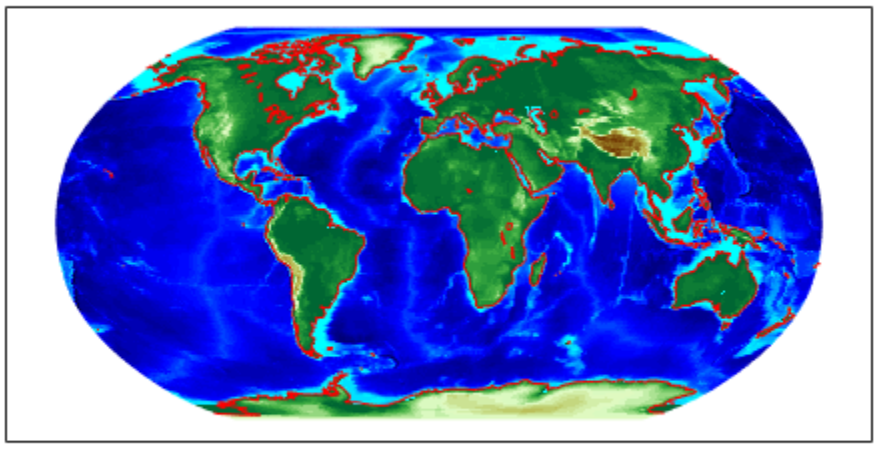
Create an `axesm`-based map using a Robinson projection. Then, display the raster and apply a colormap appropriate for elevation data.

```
axesm robinson
geoshow(topo60c,topo60cR,'DisplayType','texturemap')
demcmap(topo60c)
```



Display the vectors in red on top of the elevation map.

```
geoshow(coastlat,coastlon,'Color','r')
```



Determine and Visualize Visibility Across Terrain

You can use regular data grids of elevation data to answer questions about the mutual visibility of locations on a surface (intervisibility). For example,

- Is the line of sight from one point to another obscured by terrain?
- What area can be seen from a location?
- What area can see a given location?

The first question can be answered with the `los2` function. In its simplest form, `los2` determines the visibility between two points on the surface of a digital elevation map. You can also specify the altitudes of the observer and target points, as well as the datum with respect to which the altitudes are measured. For specialized applications, you can even control the actual and effective radius of the Earth. This allows you to assume, for example, that the Earth has a radius 1/3 larger than its actual value, a setting which is frequently used in modeling radio wave propagation.

Compute Line of Sight

The following example shows a line-of-sight calculation between two points on a regular data grid generated by the `peaks` function. The calculation is performed by the `los2` function, which returns a logical result: `1` (points are mutually visible—*intervisible*), or `0` (points are not intervisible).

1 Create an elevation grid using `peaks` with a maximum elevation of 500, and set its origin at (0°N, 0°W), with a spacing of 1000 cells per degree):

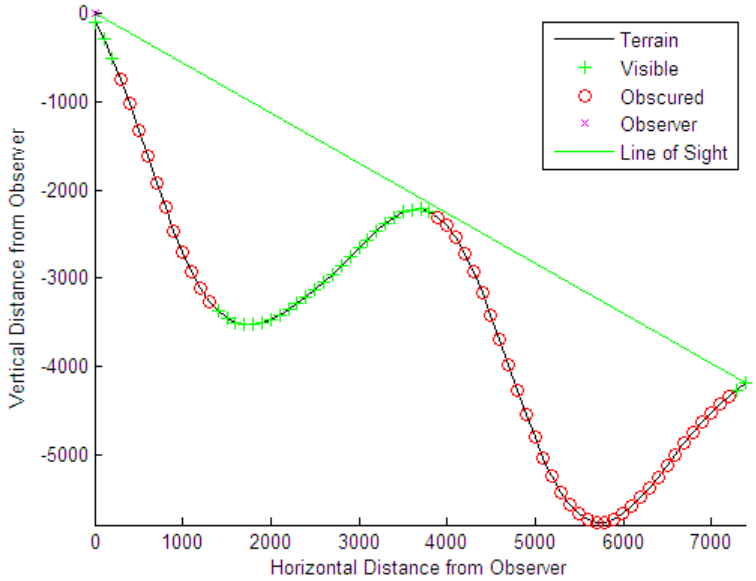
```
map = 500*peaks(100);
maplegend = [ 1000 0 0];
```

2 Define two locations on this grid to test intervisibility:

```
lat1 = -0.027;
lon1 = 0.05;
lat2 = -0.093;
lon2 = 0.042;
```

3 Calculate intervisibility. The final argument specifies the altitude (in meters) above the surface of the first location (`lat1`, `lon1`) where the observer is located (the viewpoint):

```
los2(map,maplegend,lat1,lon1,lat2,lon2,100)
ans =

     1
```



The `los2` function produces a profile diagram in a figure window showing visibility at each grid cell along the line of sight that can be used to interpret the Boolean result. In this example, the diagram shows that the line between the two locations just barely clears an intervening peak.

You can also compute the *viewshed*, a name derived from *watershed*, which indicates the elements of a terrain elevation grid that are visible from a particular location. The `viewshed` function checks for a line of sight between a fixed observer and each element in the grid. See the `viewshed` function reference page for an example.

Web Map Service and Web Maps

- “Basic WMS Terminology” on page 7-2
- “Basic Workflow for Creating WMS Maps” on page 7-3
- “Create WMS Maps When Internet Access Is Intermittent” on page 7-6
- “Explore Layers on Same Server” on page 7-9
- “Search for Layers Outside the WMS Database” on page 7-12
- “Overlay Multiple WMS Layers” on page 7-14
- “Add Legend to WMS Map” on page 7-20
- “Animate Data from WMS Layers” on page 7-24
- “Display Animation of Radar Images over GOES Backdrop” on page 7-30
- “Add Custom Basemaps from WMS Data” on page 7-32
- “Modify Your WMS Map Request” on page 7-36
- “Read WMS Maps Using Different Coordinate Reference Systems” on page 7-39
- “Read Quantitative Data from WMS Server” on page 7-43
- “Write WMS Layers to KML File” on page 7-52
- “Troubleshoot WMS Servers” on page 7-54
- “Composite and Animate Web Map Service (WMS) Meteorological Layers” on page 7-58
- “Web Map Migration Strategies” on page 7-70
- “Use Base Layers in Geographic Axes” on page 7-75
- “Add Interactive Controls to Geographic Axes” on page 7-80

Basic WMS Terminology

- **Open Geospatial Consortium, Inc. (OGC)** — An organization comprising companies, government agencies, and universities that defines specifications for providers of geospatial data and developers of software designed to access that data. The specifications ensure that providers and clients can talk to each other and thus promote the sharing of geospatial data worldwide. You can access the Web Map Server Implementation Specification at the OGC website.
- **Web Map Service** — The OGC® defines a Web Map Service (WMS) as an entity that "produces maps of spatially referenced data dynamically from geographic information."
- **WMS server**— A server that follows the guidelines of the OGC to render maps and return them to clients.
- **georeferenced** — Tied to a specific location on the Earth.
- **raster data** — Data represented as a matrix in which each element corresponds to a specific rectangular or quadrangular geographic area.
- **map** — The OGC defines a map as "a portrayal of geographic information as a digital image file suitable for display on a computer screen."
- **raster map** — Geographically referenced information stored as a regular array of cells.
- **layer** — A data set containing a specific type of geographic information. Information can include temperature, elevation, weather, orthoimagery, boundaries, demographics, topography, transportation, environmental measurements, or various data from satellites.
- **capabilities document** — An XML document containing metadata describing the geographic content offered by a server.

See Also

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Basic Workflow for Creating WMS Maps

A Web Map Service (WMS) provides images of publicly accessible geospatial information from web-based sources. This example shows how to find and display a WMS map of satellite imagery for a region around Europe.

Search WMS Database

A layer is a data set that contains a specific type of geographic information, such as elevation, weather, or orthoimagery. Mapping Toolbox™ contains a database, called the *WMS Database*, that includes more than 100,000 layers from more than 1000 servers. You can search the WMS Database by using the `wmsfind` function. By default, the `wmsfind` function searches the WMS Database for matching layer names and layer titles.

For this example, search the WMS Database for layers that mention `eox`. For more information about EOX::Maps, see EOX::Maps.

```
eox = wmsfind("eox");
```

The `wmsfind` function returns layers as `WMSLayer` objects. In this case, the function returned multiple layers. Note that your results might be different because the WMS Database changes each release.

Refine Search

Refine your search based on the WMS Database by using the `refine` function or based on geographic limits by using the `refineLimits` function. If your original search provides only one layer, then you do not need to refine your search.

For this example, refine your search to find layers in the WMS Database that also contain blue marble imagery from the NASA Earth Observatory.

```
eox_marble = refine(eox,"blue marble");
```

In this case, there are multiple layers in the WMS Database from EOX::Maps that contain blue marble imagery. Refine your search again to find layers with valid latitude limits.

```
eox_marble_limits = refineLimits(eox_marble,Latlim=[-90 90]);
```

The refined search includes one layer.

Synchronize Layer with Server

Get up-to-date information about the layer by synchronizing it with the web server. The `wmsupdate` function updates the properties of `WMSLayer` objects, including the `Abstract`, `CoordRefSysCodes`, and `Details` properties.

Update the layer.

```
eox_update = wmsupdate(eox_marble_limits);
```

Read and Display Map

Read the WMS map from the server by using the `wmsread` function. You can customize the map by specifying properties such as the geographic limits, image dimensions, and background color.

Get the geographic limits for an area of interest (AOI) surrounding Europe. Find the latitude and longitude limits of the AOI by using the `bounds` function.

```
europe = geocode("Europe");
[latlim,lonlim] = bounds(europe.Shape);
```

Using the limits of the AOI, read the layer as an array and a `GeographicCellsReference` object. The `GeographicCellsReference` object ties the map to a specific location on Earth.

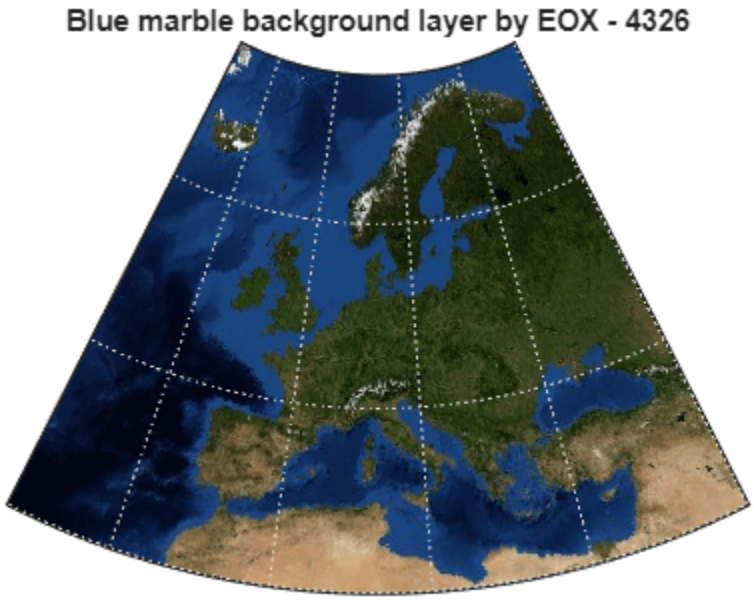
```
[A,R] = wmsread(eox_update,Latlim=latlim,Lonlim=lonlim);
```

Create an `axesm`-based map using the limits of the AOI. Display the layer on the map. Then, customize the map by removing the parallel and meridian labels.

```
figure
worldmap(latlim,lonlim)
geoshow(A,R)
plabel off
mlabel off
```

Add a title by specifying the layer title.

```
title(eox_update.LayerTitle)
```



See Also

Functions
`wmsfind` | `wmsupdate` | `wmsread`

Objects
`WMSLayer` | `GeographicCellsReference`

More About

- “Basic WMS Terminology” on page 7-2
- “Modify Your WMS Map Request” on page 7-36

Create WMS Maps When Internet Access Is Intermittent

Reading data from WMS servers requires Internet access. However, once you read data from a server, you can save the data as a MAT or GeoTIFF file. Then, you can load the data without reading it again from the server. Saving WMS data to files is useful when you have intermittent Internet access, or when you want to share data with someone without Internet access.

This example shows how to create a WMS map for an area around Boston. The behavior of the example depends on whether you have Internet access, but the resulting map is the same.

Throughout the example, specify whether you have Internet access by using the logical scalar `useInternet`.

- If `useInternet` is `true`, read data from the WMS server and update the saved files.
- If `useInternet` is `false`, read data from the saved files.

```
useInternet = false;
```

If you use this example as a model to create your own WMS maps, you must create the saved files the first time you run the example by setting `useInternet` to `true`.

Search the WMS Database for basemap layers from the USGS National Map [1] on page 7-7. Refine the search to find a layer containing only orthoimagery. This step does not require Internet access, because the WMS Database is installed with Mapping Toolbox™.

```
bmapLayers = wmsfind("basemap.nationalmap",SearchFields="ServerURL");
usgsImageryLayer = refine(bmapLayers,"USGSImageryOnly",SearchFields="ServerURL");
```

Synchronize the layer with the server. The `wmsupdate` function adds an abstract, the attribute and style information, and the coordinate reference system information to the layer.

- If `useInternet` is `true`, synchronize the layer with the server. Save the layer to a MAT file.
- If `useInternet` is `false`, load the MAT file.

```
if useInternet
    usgsImageryLayer = wmsupdate(usgsImageryLayer);
    save("usgsImageryLayer.mat","usgsImageryLayer")
else
    load usgsImageryLayer.mat
end
```

Specify the geographic limits and the image size. For this example, specify the limits for a region around Boston and a square image with a side length of 750 pixels.

```
latlim = [42.3453 42.3711];
lonlim = [-71.099 -71.0454];
s = 750;
```

Read the image from the server.

- If `useInternet` is `true`, then read the map as an array and a `GeographicCellsReference` object. Save the array and reference object to a GeoTIFF file.
- If `useInternet` is `false`, then read the GeoTIFF file.

```
if useInternet
    [A,R] = wmsread(usgsImageryLayer,Latlim=latlim,Lonlim=lonlim, ...
        ImageHeight=s,ImageWidth=s);
    geotiffwrite("usgsImageryLayer.tif",A,R)
else
    [A,R] = readgeoraster("usgsImageryLayer.tif");
end
```

Create a map with appropriate latitude and longitude limits for the data. Display the map.

```
figure
usamap(A,R)
geoshow(A,R)
```

[1] Data available from U.S. Geological Survey, National Geospatial Program.

See Also

Functions
`wmsread` | `wmsupdate`

Objects
`GeographicCellsReference` | `WMSLayer`

Related Examples

- “Basic Workflow for Creating WMS Maps” on page 7-3
- “Proxy Server Authentication”

- “Use Basemaps in Offline Environments” on page 4-64
- “Display Data over Basemaps When Internet Access Is Intermittent” on page 4-79

Explore Layers on Same Server

You might find a layer you like in the WMS Database and want to find other layers on the same server. You can explore other layers by using a capabilities document.

Specify the URL of the NASA SVS Image Server. Use the `wmsinfo` function to return the contents of the capabilities document as a `WMSCapabilities` object. A *capabilities document* is an XML document containing metadata that describes the geographic content a server offers.

```
serverURL = "http://svs.gsfc.nasa.gov/cgi-bin/wms?";
capabilities = wmsinfo(serverURL);
```

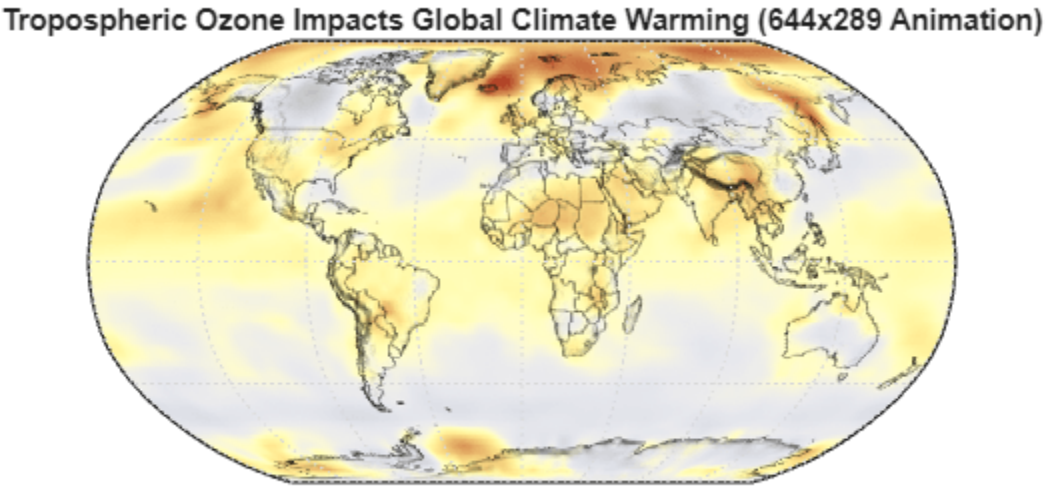
Set the layer titles to a variable. View the first three titles.

```
layerTitles = {capabilities.Layer.LayerTitle}';
layerTitles(1:3)
```

```
ans = 3×1 cell
    {'African Fires During 2002 (1024x1024 Animation)'                 }
    {'Background Image for African Fires During 2002 (WMS)'            }
    {'Aurora over the North Pole on April 17, 1999 (360x100 Animation)'}
```

Read the layer containing tropospheric ozone impacts. Display the map.

```
layerTitle = "Tropospheric Ozone Impacts Global Climate Warming";
layer = refine(capabilities.Layer,layerTitle);
[A,R] = wmsread(layer);

figure
worldmap(A,R)
geoshow(A,R)
title(layer.LayerTitle)
plabel off
mlabel off
```



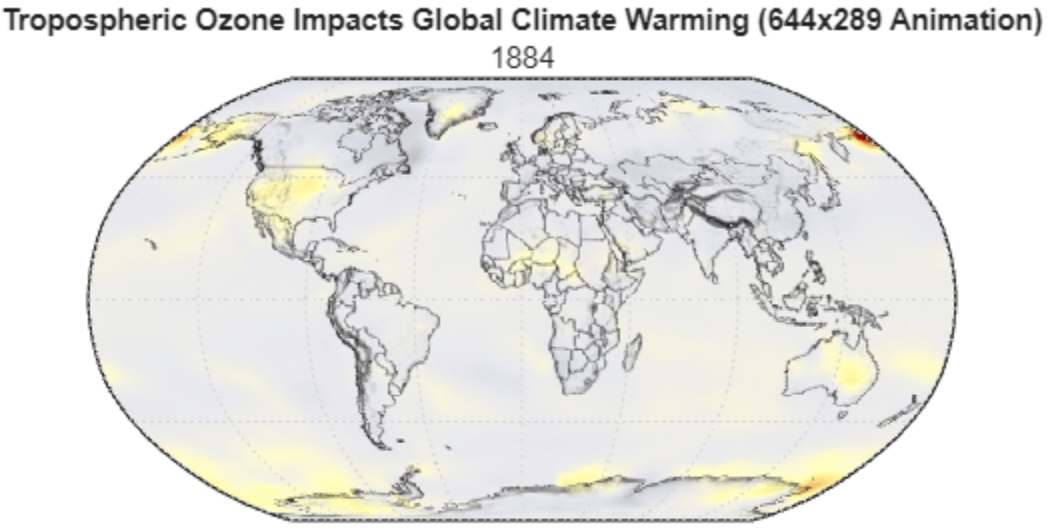
This layer contains data from different years. You can examine the available data by accessing the `layer.Details.Dimension` structure.

```
layer.Details.Dimension
```

```
ans = struct with fields:
              Name: 'time'
             Units: 'ISO8601'
        UnitSymbol: ''
           Default: '1994-'
    MultipleValues: 0
      NearestValue: 0
           Current: 0
            Extent: '1884-/1994-/P1Y'
```

Display the map for the year 1884 and compare it with the map for 1994, the default year (displayed previously).

```
year = "1884";
[A2,R] = wmsread(layer,Time=year);
figure
worldmap(A2,R)
geoshow(A2,R)
title(layer.LayerTitle,year)
plabel off
mlabel off
```



See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Search for Layers Outside the WMS Database

Search for layers using a web browser instead of using the WMS Database.

Select a search engine. If you are using Google®, select **Images** and enter `getmap wms` into the search box.

Choose a map by viewing the images. Click the map link and find the WMS `GetCapabilities` request somewhere on the page. If you cannot find a `GetCapabilities` request, try another map.

For example, this code shows the URL of a WMS `GetCapabilities` request from the USGS National Map.

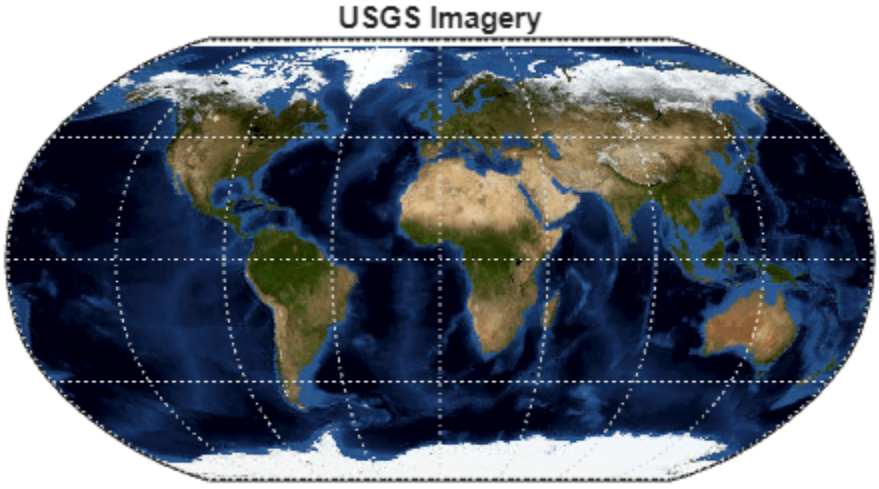
```
url = "https://basemap.nationalmap.gov/ArcGIS/services/USGSImageryOnly/" + ...
       "MapServer/WMSServer?REQUEST=GetCapabilities&VERSION=1.3.0&SERVICE=WMS";
```

Return the capabilities document by using the `wmsinfo` function.

```
c = wmsinfo(url);
```

Read a layer and display it on a world map.

```
layer = c.Layer(1);
[A,R] = wmsread(layer);

figure
worldmap world
geoshow(A,R)
mlabel off
plabel off
title("USGS Imagery")
```



See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Overlay Multiple WMS Layers

In this section...

“Create Composite Map of Multiple Layers from One Server” on page 7-14
“Combine Layers from One Server with Data from Other Sources” on page 7-15
“Drape Orthoimagery Over DEM” on page 7-17

Create Composite Map of Multiple Layers from One Server

The WMS specification enables a server to merge multiple layers into a single raster map. The EOX::Maps server contains several data layers, such as coastlines, hydrography, imagery, and terrain. Read and display a composite of multiple layers from the EOX::Maps server. For more information about EOX::Maps, see EOX::Maps.

Read the capabilities document from the EOX::Maps server. Extract the list of layers from the capabilities document.

```
info = wmsinfo("https://tiles.maps.eox.at/wms?");
eox = info.Layer;
```

Create an array of multiple layers that includes terrain, coastlines, and hydrography.

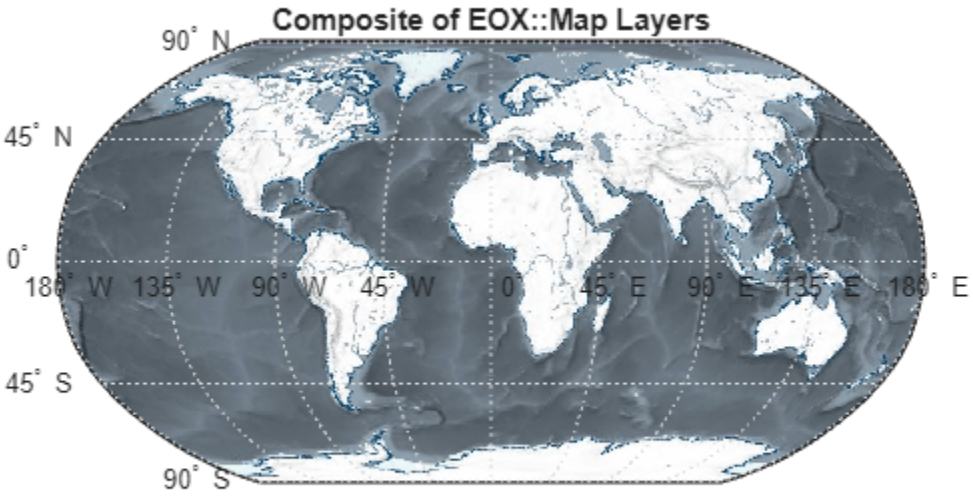
```
terrain = refine(eox,"terrain-light",MatchType="exact");
coast = refine(eox,"coastline",MatchType="exact");
hydro = refine(eox,"hydrography",MatchType="exact");

layers = [hydro; coast; terrain];
```

Read the composite map from the server. Get a map that uses a cell size of 0.5 degrees by specifying the height and width of the image. Use a transparent value for pixels that do not represent data values.

```
[A,R] = wmsread(layers,Transparent=true,ImageHeight=360,ImageWidth=720);
```

Display the composite map.

```
figure
worldmap world
geoshow(A,R)
title("Composite of EOX::Map Layers")
```



Combine Layers from One Server with Data from Other Sources

This example shows how to merge raster data from a WMS server with vector data from shapefiles.

Create a geographic raster reference object by using the `georefcells` function. Specify latitude and longitude limits that cover the globe. Specify the cell extent for the raster as 0.5 degrees. For consistency with the `wmsread` function, specify columns that run from north to south.

```
latlim = [-90 90];
lonlim = [-180 180];
cellExtent = 0.5;
R = georefcells(latlim,lonlim,cellExtent,cellExtent,ColumnsStartFrom="north");
```

Read world land areas from a shapefile into the workspace. The shapefile represents the land areas using polygons. Convert the polygons into a raster map. Specify the size of the raster map using the size data embedded in the reference object.

```
land = shaperead("landareas.shp",UseGeoCoords=true);
latAreas = [land.Lat];
lonAreas = [land.Lon];
land = vec2mtx(latAreas,lonAreas,zeros(R.RasterSize),R,"filled");
```

Read world rivers from a shapefile into the workspace. The shapefile represents the rivers using polylines. Convert the polylines into a raster map.

```
rivers = shaperead("worldrivers.shp",UseGeoCoords=true);
latRivers = [rivers.Lat];
lonRivers = [rivers.Lon];
rivers = vec2mtx(latRivers,lonRivers,land,R);
```

Create a new raster map by merging the land data with the rivers data.

```
A = land;
A(rivers == 1) = 3;
```

The land data and rivers data use the same coordinate reference system (CRS). Get the CRS by using the `shapeinfo` function. Then, set the CRS of the reference object to match the CRS of the land and rivers data.

```
info = shapeinfo("landareas.shp");
R.GeographicCRS = info.CoordinateReferenceSystem;
```

Get a layer containing boundaries data from the EOC Basemap Map Service server.

```
info = wmsinfo("https://geoservice.dlr.de/eoc/basemap/wms?");
eoc = info.Layer;
layer = refine(eoc,"ne_countries");
```

Read the data from the server. Specify the size of the image using the size data embedded in the reference object.

```
height = R.RasterSize(1);
width  = R.RasterSize(2);
[B,BR] = wmsread(layer,ImageFormat="image/png",ImageHeight=height,ImageWidth=width);
```

Confirm that the reference object for the boundaries data is the same as the reference object for the merged land and rivers data.

```
isequal(BR,R)
```

```
ans = logical
   1
```

Merge the boundaries data with the land and rivers data.

```
index = B(:,:,1) ~= 255 & B(:,:,2) ~= 255 & B(:,:,3) ~= 255;
A(index) = 1;
```

Display the result on a map.

```
figure
worldmap(A,R)
geoshow(A,R,DisplayType="texturemap")
```

Apply a colormap that uses green for the land areas, black for the boundaries, blue for the rivers, and light blue for the ocean.

```
landColor = [0.45 0.60 0.30];
boundaryColor = [0 0 0];
riverColor = [0 0.5 1];
oceanColor = [0 0 1];
colormap([landColor; boundaryColor; riverColor; oceanColor])
```

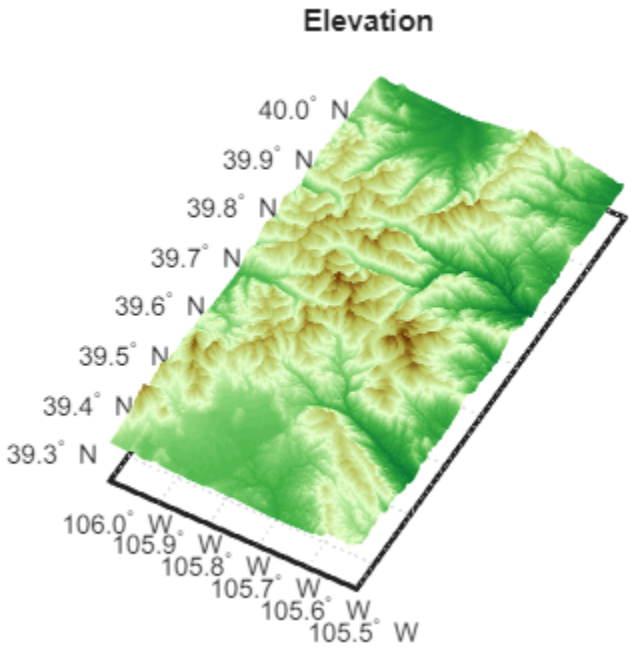
Drape Orthoimagery Over DEM

Read elevation data and a geographic postings reference for an area around South Boulder Peak in Colorado. Crop the elevation data to a smaller area using the `geocrop` function.

```
[fullZ,fullR] = readgeoraster("n39_w106_3arc_v2.dt1",OutputType="double");

latlim = [39.25 40.0];
lonlim = [-106 -105.5];
[Z,R] = geocrop(fullZ,fullR,latlim,lonlim);
```

Display the elevation data. To do this, create an `axesm`-based map for the United States, plot the data as a surface, and apply an appropriate colormap. View the map in 3-D by adjusting the camera position and target. Set the vertical exaggeration by using the `daspectm` function.

```
figure
usamap(R.LatitudeLimits,R.LongitudeLimits)
geoshow(Z,R,DisplayType="surface")
demcmap(Z)
title("Elevation");

cameraPosition = [218100 4367600 183700];
cameraTarget = [0 4754200 2500];
set(gca,"CameraPosition",cameraPosition, ...
        "CameraTarget",cameraTarget)
daspectm("m",3)
```



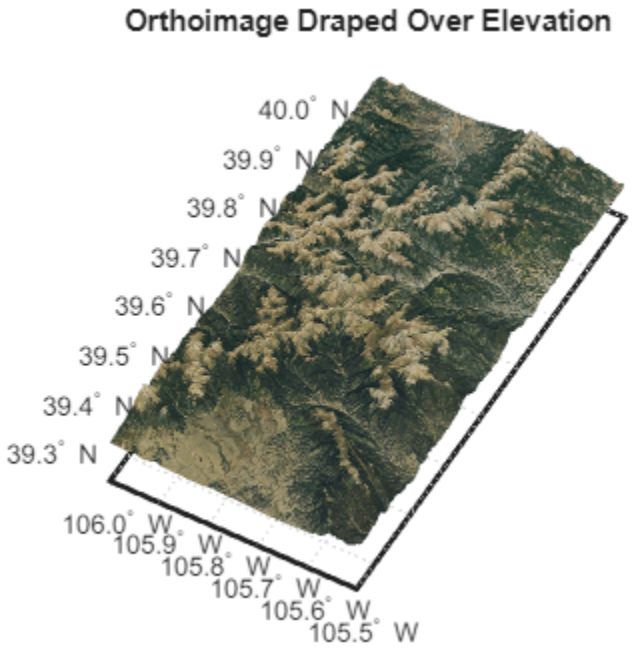
Drape an orthoimage over the elevation data. To do this, first get the names of high-resolution orthoimagery layers from the USGS National Map using the `wmsinfo` function. In this case, the orthoimagery layer is the only layer from the server. Use multiple attempts to connect to the server in case it is busy.

```
numberOfAttempts = 5;
attempt = 0;
info = [];
serverURL = ...
   "http://basemap.nationalmap.gov/ArcGIS/services/USGSImageryOnly/MapServer/WMSServer?";
while(isempty(info))
    try
        info = wmsinfo(serverURL);
        orthoLayer = info.Layer(1);
    catch e
        attempt = attempt + 1;
        if attempt > numberOfAttempts
            throw(e);
        else
            fprintf('Attempting to connect to server:\n"%s"\n',serverURL)
        end
    end
end
```

Request a map of the orthoimagery layer using the `wmsread` function. To display the orthoimagery, use the `geoshow` function and set the `CData` property to the layer.

```
imageHeight = size(Z,1);
imageWidth  = size(Z,2);

orthoImage = wmsread(orthoLayer,Latlim=R.LatitudeLimits, ...
    Lonlim=R.LongitudeLimits,ImageHeight=imageHeight, ...
    ImageWidth=imageWidth);
```

```
figure
usamap(R.LatitudeLimits,R.LongitudeLimits)
geoshow(Z,R,DisplayType="surface",CData=orthoImage);
title("Orthoimage Draped Over Elevation");
set(gca,"CameraPosition",cameraPosition, ...
        "CameraTarget",cameraTarget)
daspectm('m',3)
```



The DTED file used in this example is from the US Geological Survey.

See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Add Legend to WMS Map

WMS servers render layers as images. About 45% of WMS servers in the WMS Database provide legends to help you interpret the pixel colors of the images. You can download the legend image for a layer by accessing the `Details.Style.LegendURL` field of the `WMSLayer` object.

This example shows how to read and display a WMS map of surface temperatures, download the associated legend image, and combine the WMS map and legend into a single image.

Read and Display Map

Search the WMS Database for layers from the NASA Goddard Space Flight SVS Image Server, then find the layer that contains urban temperature signatures. Synchronize the layer with the server by using the `wmsupdate` function. To access the legend image, you must synchronize the layer with the server.

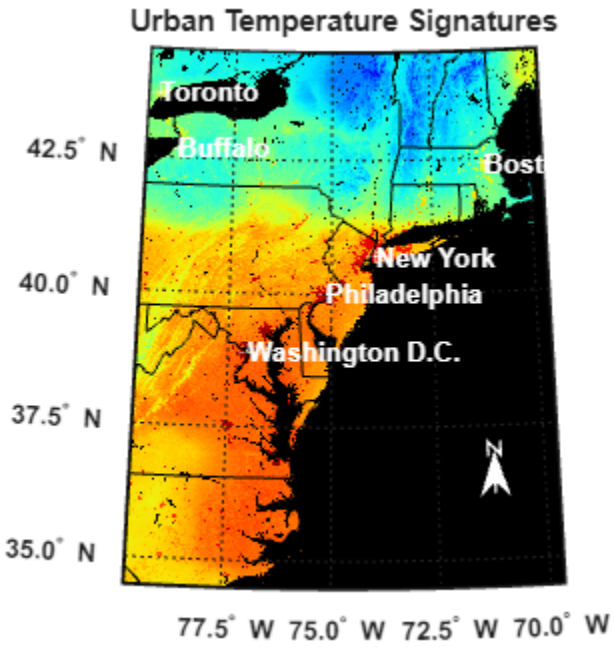
```
layers = wmsfind("svs.gsfc.nasa.gov",SearchField="serverurl");
urban_temperature = refine(layers,"urban*temperature");
urban_temperature = wmsupdate(urban_temperature);
```

Read and display the layer on a map.

```
[A,R] = wmsread(urban_temperature);
mapFigure = figure;
usamap(A,R)
geoshow(A,R)
```

Customize the map by adding city markers, state boundaries, meridian and parallel labels, a north arrow, and a title.

```
latlim = R.LatitudeLimits;
lonlim = R.LongitudeLimits;

GT = readgeotable("worldcities.shp");
cities = geoclip(GT.Shape,latlim,lonlim);
GT = GT(cities.NumPoints ~= 0,:);

geoshow(GT,MarkerEdgeColor='w',Color="w")
for k=1:height(GT)
    lat = GT(k,:).Shape.Latitude;
    lon = GT(k,:).Shape.Longitude;
    n = GT(k,:).Name;
    textm(lat,lon,n,Color="w",FontWeight="bold")
end

geoshow("usastatehi.shp",FaceColor="none",EdgeColor="black")

mlabel(FontWeight="bold")
plabel(FontWeight="bold")

northarrow(Facecolor="w",EdgeColor="w",Latitude=36.249,Longitude=-71.173)

title("Urban Temperature Signatures")
```



Search for Legend

Determine if a legend image is available by accessing the `Details.Style.LegendURL` field of the layer.

```
urban_temperature.Details.Style.LegendURL
```

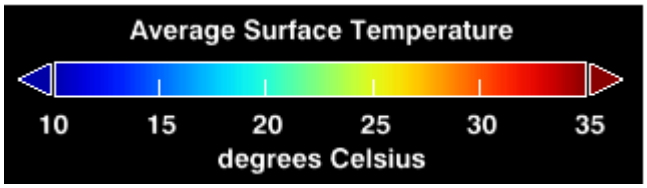
```
ans = struct with fields:
    OnlineResource: 'http://svs.gsfc.nasa.gov/vis/a000000/a003100/a003152/temp_bar.png'
            Format: 'image/png'
            Height: 90
             Width: 320
```

The fields are not empty, so a legend is available. Download the legend image.

```
url = urban_temperature.Details.Style.LegendURL.OnlineResource;
legendImage = webread(url);
```

Create a figure and axes that are the same size as the image. Then, display the image. Note that the legend is an image of a colorbar, not a `Legend` object.

```
sz = size(legendImage);

legendFigure = figure;
pos = legendFigure.Position;
legendFigure.Position = [pos(1) pos(2) sz(2) sz(1)];

ax = gca;
ax.Units = "pixels";
ax.Position = [0 0 sz(2) sz(1)];

imshow(legendImage)
```



Combine Map and Legend

When you display images using the `geoshow` function, the map projection might cause text to appear warped. Instead, convert the map into an image and combine the map image with the legend image.

Convert the map into an image by using the `getframe` and `frame2im` functions.

```
mapImage = getframe(mapFigure);
mapImage = frame2im(mapImage);
```

To combine the images, the width of the images must be equal. In this example, the width of the legend image is smaller than the width of the map image. Make the widths the same by padding the legend image.

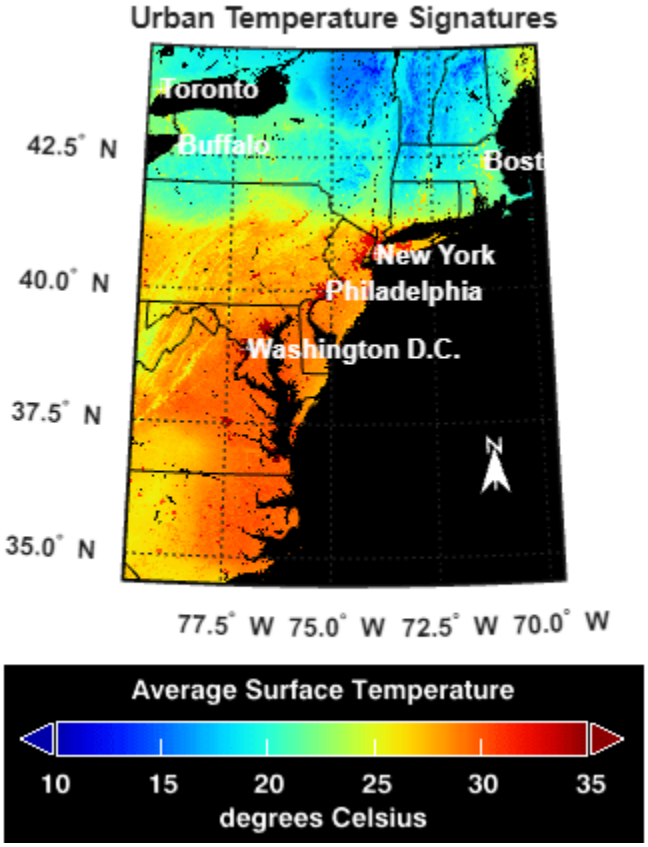
```
mapWidth = size(mapImage,2);
legendImage = paddata(legendImage,mapWidth,Dimension=2,Side="both",FillValue=255);
```

Combine the images.

```
combinedImage = [mapImage; legendImage];
```

Create a figure and axes that are the same size as the combined image. Display the combined image in the figure.

```
combinedsz = size(combinedImage);

combinedFigure = figure;
pos = combinedFigure.Position;
combinedFigure.Position = ...
    [pos(1) pos(2) combinedsz(2) combinedsz(1)];

ax = gca;
ax.Units = "pixels";
ax.Position = [0 0 combinedsz(2) combinedsz(1)];

imshow(combinedImage)
```



See Also

Functions
`wmsfind` | `wmsupdate` | `wmsread`

Objects
`WMSLayer`

Animate Data from WMS Layers

In this section...
“Create Movie of Terra/MODIS Maps” on page 7-24
“Create Animated GIF of WMS Maps” on page 7-26
“Create Animation of Time-Lapse Radar Observations” on page 7-27

Create Movie of Terra/MODIS Maps

You can create maps of the same geographic region at different times and view them as a movie. For a period of seven days, read and display a daily composite of visual images from NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) scenes captured during the month of December 2010.

Search the WMS Database for a MODIS layer from the NASA Earth Observations (NEO) server.

```
neo = wmsfind("neo*nasa","SearchFields","serverurl");
modis = refine(neo,"true*color*terra*modis");
modis = wmsupdate(modis);
```

Create a `WebMapServer` object and a `WMSMapRequest` object.

```
server = WebMapServer(modis.ServerURL);
mapRequest = WMSMapRequest(modis,server);
```

The `Extent` field provides information about how to retrieve individual frames. For this extent, you can request a single day because the extent is defined by day (`'/P1D'`). Because this layer updates periodically, the value of your `Extent` field might be different.

```
modis.Details.Dimension.Extent(1:25)
```

```
ans =
'2006-09-01/2006-09-14/P1D'
```

Create an array indicating the first seven days. Set the start time to December 10, 2022 and use a serial date number.

```
days = 1:7;
time = "2022-12-10";
startTime = datenum(time);
```

Open a figure window with axes appropriate for the region specified by the MODIS layer.

```
hFig = figure;
worldmap(mapRequest.Latlim,mapRequest.Lonlim);
```

Save each frame to a video file.

```
videoFilename = "modis_dec.avi";
writer = VideoWriter(videoFilename);
writer.FrameRate = 1;
writer.Quality = 100;
writer.open;
```

Set the `Visible` property of the figure to `'off'` to hide the figure while populating the frames. For each day, read a map of the MODIS layer. Set the `Time` property to the day number. Ignore data not found on the server or errors issued by the server by using a `try/catch` statement. For correct indexing, set the start time to one day less.

```
hFig.Visible = "off";
startTime = startTime - 1;
for k = days
    try
        mapRequest.Time = startTime + k;
        timeStr = datestr(mapRequest.Time);
        dailyImage = getMap(server,mapRequest.RequestURL);
        geoshow(dailyImage,mapRequest.RasterReference);
        title({mapRequest.Layer.LayerTitle,timeStr}, ...
            "Interpreter","none","FontWeight","bold")
        frame = getframe(hFig);
        writer.writeVideo(frame);
    catch e
        fprintf(['Server error: %s.\n', ...
            'Ignoring frame number %d on day %s.\n'], ...
            e.message,k,timeStr)
    end
end
writer.close
```

Read in all video frames.

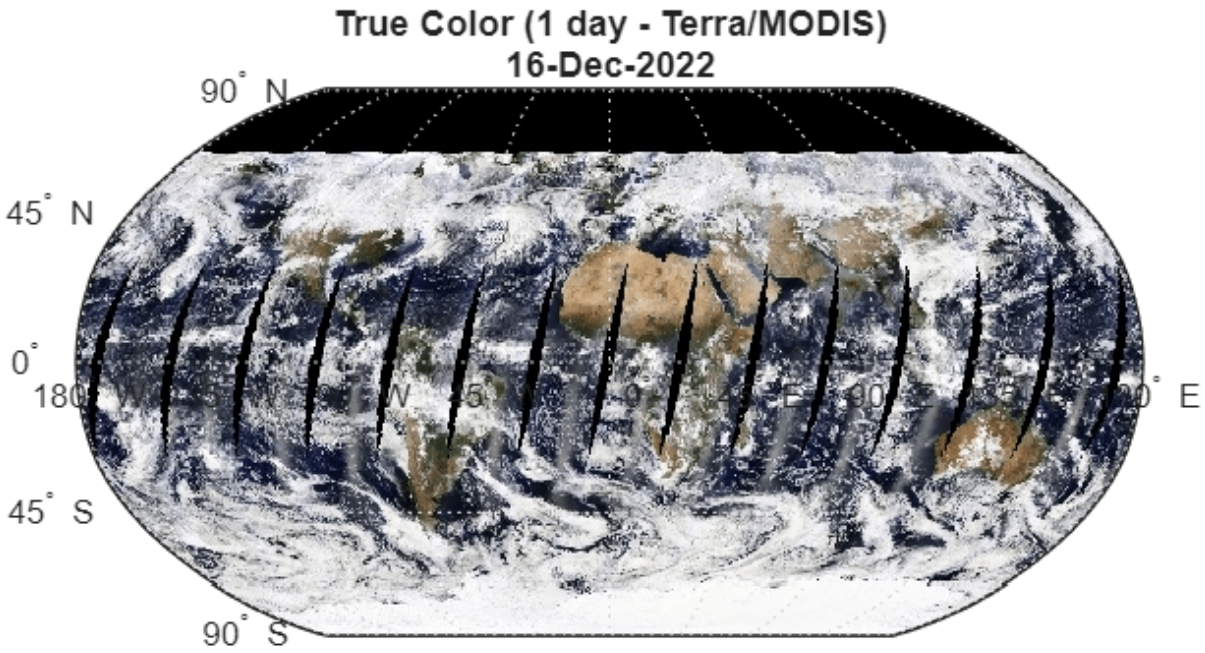
```
v = VideoReader(videoFilename);
vidFrames = read(v);
numFrames = get(v,"NumFrames");
```

Create a movie structure from the video frames.

```
frames = struct("cdata",[],"colormap",[]);
frames(numFrames) = frames(1);
for k = 1 : numFrames
    frames(k).cdata = vidFrames(:,:,:,k);
    frames(k).colormap = [];
end
```

Set the `Visible` property of the figure to `'on'` and play the movie once at the frame rate of the video.

```
hFig.Visible = "on";
movie(hFig,frames,1,v.FrameRate)
```



Create Animated GIF of WMS Maps

Read and display an animation of the Larsen Ice Shelf experiencing a dramatic collapse between January 31, 2002 and March 7, 2002.

Search the WMS Database for the phrase `'Larsen Ice Shelf'`.

```
iceLayer = wmsfind('Larsen Ice Shelf');
```

Create a `WebMapServer` by specifying the server URL of the first layer. Synchronize the layer with the server by using the `updateLayers` function.

```
server = WebMapServer(iceLayer(1).ServerURL);
iceLayer = updateLayers(server,iceLayer(1));
```

Create a `WMSMapRequest` object.

```
request = WMSMapRequest(iceLayer(1),server);
```

The `Extent` field provides the available values for a dimension, in this case time. Set the value of the extent to a variable. Calculate the number of required frames.

```
extent = string(iceLayer.Details.Dimension.Extent);
extent = split(extent,",");
numFrames = numel(extent);
```

Create a figure and set up a map with appropriate geographic limits. Customize the appearance of the parallels and meridians.

```
f = figure;
worldmap(request.Latlim,request.Lonlim)
setm(gca,'MLineLocation',1,'MLabelLocation',1, ...
   'MLabelParallel',-67.5,'LabelRotation','off');
```

Initialize the value of `animated` to `0`.

```
animated(1,1,1,numFrames) = 0;
```

Set the `Visible` property of the figure to `'off'` to hide the map while populating the frames. Read the image of the Larsen Ice Shelf on different days.

```
f.Visible = 'off';
for k=1:numFrames
   request.Time = extent(k);
   iceImage = getMap(server,request.RequestURL);
   geoshow(iceImage,request.RasterReference)
   title(request.Time,'Interpreter','none')
   frame = getframe(f);
   if k == 1
      [animated, cmap] = rgb2ind(frame.cdata,256,'nodither');
   else
      animated(:,:,1,k) = rgb2ind(frame.cdata,cmap,'nodither');
   end
   pause(2)
end
```

Save the frames as an animated GIF and view the GIF in a browser.

```
filename = 'wmsanimated.gif';
imwrite(animated,cmap,filename,'DelayTime',1.5, ...
   'LoopCount',inf);
web(filename)
```

Create Animation of Time-Lapse Radar Observations

Display Next-Generation Radar (NEXRAD) images for the United States using data from the Iowa Environmental Mesonet (IEM) web map server. The server stores layers covering the past 50 minutes up to the present time in increments of 5 minutes. Read and display the merged layers.

Find layers in the WMS Database that include `'mesonet'` and `'nexrad'` in their `ServerURL` fields.

```
mesonet = wmsfind("mesonet*nexrad",SearchField="serverurl");
```

NEXRAD Base Reflect Current (`"nexrad-n0r"`) measures the intensity of precipitation. Refine your search to include only layers with this phrase in one of the search fields.

```
nexrad = refine(mesonet,"nexrad-n0r",SearchField="any");
```

Remove the 900913 layers because they are intended for Google Maps™ overlay. Remove the WMST layer because it contains data for different times.

```
layers_900913 = refine(nexrad,"900913",SearchField="layername");
layer_wmst = refine(nexrad,"wmst",SearchField="layername");
rmLayerNames = {layers_900913.LayerName layer_wmst.LayerName};
index = ismember({nexrad.LayerName}, rmLayerNames);
nexrad = nexrad(~index);
```

Update the `nexrad` layer to fill in all fields and obtain the most recent data.

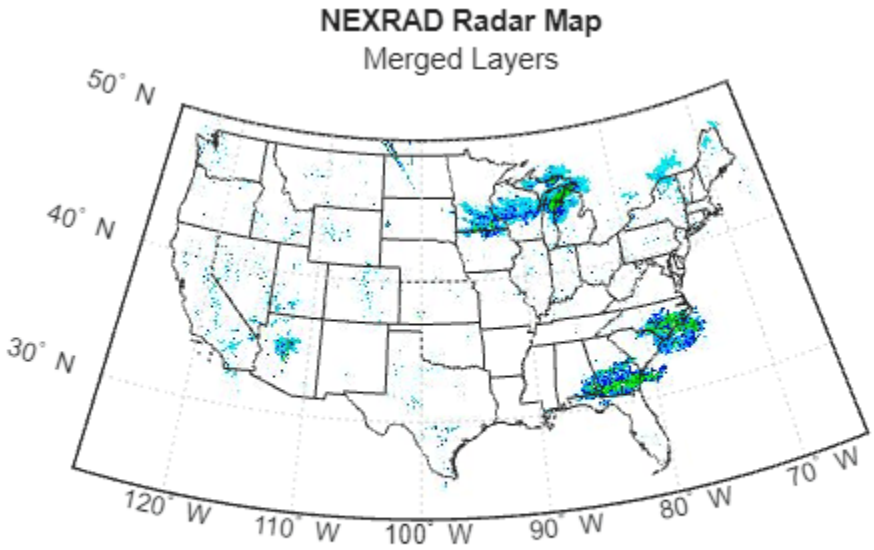
```
nexrad = wmsupdate(nexrad,AllowMultipleServers=true);
```

Create a map of the conterminous United States. Get the latitude and longitude limits of the map.

```
figure
usamap("conus")
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
```

Read and display the merged layers.

```
[A,R] = wmsread(nexrad,Latlim=latlim,Lonlim=lonlim);
geoshow(A,R)
geoshow("usastatehi.shp",FaceColor="none")
title("NEXRAD Radar Map","Merged Layers")
```



Loop through the sequence of time-lapse radar observations. Set the `Visible` property of the figure to `'off'` to hide the figure while populating the frames.

```
hfig = figure(Visible="off");
usamap("conus")
hstates = geoshow("usastatehi.shp",FaceColor="none");
numFrames = numel(nexrad);
frames = struct("cdata",[],"colormap",[]);
frames(numFrames) = frames;
```

```
hmap = [];
frameIndex = 0;
for k = numFrames:-1:1
   frameIndex = frameIndex + 1;
   delete(hmap)
   [A, R] = wmsread(nexrad(k),Latlim=latlim,Lonlim=lonlim);
   hmap = geoshow(A,R);
   uistack(hstates,"top")
   title(nexrad(k).LayerName)
   frames(frameIndex) = getframe(hfig);
end
```

Create an array to write out as an animated GIF.

```
animated(1,1,1,numFrames) = 0;
for k=1:numFrames
   if k == 1
      [animated,cmap] = rgb2ind(frames(k).cdata,256,"nodither");
   else
      animated(:,:,1,k) = ...
         rgb2ind(frames(k).cdata,cmap,"nodither");
   end
end
```

Save the animated GIF and view it in a browser.

```
filename = "wmsnexrad.gif";
imwrite(animated,cmap,filename,DelayTime=1.5,LoopCount=inf);
web(filename)
```

See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Display Animation of Radar Images over GOES Backdrop

This example shows how to display NEXRAD radar images. The images cover the past 24 hours, sampled at one-hour intervals, for the United States using data from the IEM WMS server.

Search the WMS Database for the `'nexrad-n0r-wmst'` layer. Synchronize the layer with the server.

```
wmst = wmsfind('nexrad-n0r-wmst','SearchField','layername');
wmst = wmsupdate(wmst);
```

Search the WMS Database for a generated CONUS composite of GOES IR imagery. Synchronize the layer with the server.

```
goes = wmsfind('goes*conus*ir','SearchField','layername');
goes = wmsupdate(goes);
```

Create a map of the conterminous United States.

```
hfig = figure;
region = 'conus';
usamap(region)
borders = geoshow('usastatehi.shp','FaceColor','none');
```

Get the latitude and longitude limits of the map.

```
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
```

Read the GOES layer to use as a backdrop image.

```
cellsize = 0.1;
[backdrop, R] = wmsread(goes,'ImageFormat','image/png', ...
   'Latlim',latlim,'Lonlim',lonlim,'Cellsize',cellsize);
```

Find the current time minus 24 hours. Set up frames to hold the data from `getframe`.

```
now_m24 = datestr(now-1);
hour_m24 = [now_m24(1:end-5) '00:00'];
hour = datenum(hour_m24);
hmap = [];
numFrames = 24;
frames = struct('cdata',[],'colormap',[]);
frames(numFrames) = frames;
```

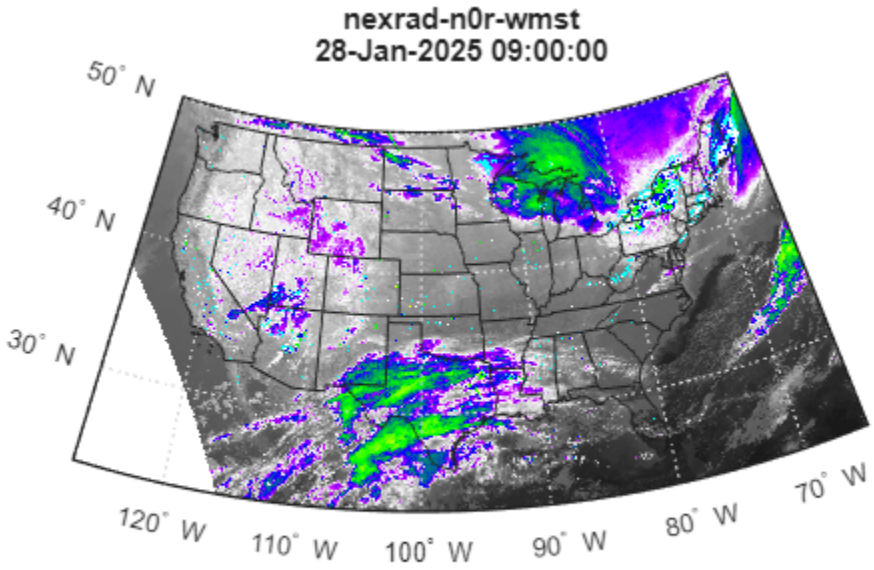
Set the `Visible` property of the figure to `'off'` to hide the figure while populating the frames. For each hour, obtain the hourly NEXRAD map data and combine it with a copy of the backdrop. Because of how this Web server handles PNG format, the resulting map data has an image with class `double`. Thus, you must convert it to `uint8` before merging.

```
hfig.Visible = 'off';
for k=1:numFrames
    time = datestr(hour);
    [A,R] = wmsread(wmst,'Latlim',latlim,'Lonlim',lonlim, ...
        'Time',time,'CellSize',cellsize, ...
        'BackgroundColor',[0 0 0],'ImageFormat','image/png');
```

```
    delete(hmap)
    index = any(A > 0, 3);
    combination = backdrop;
    index = cat(3,index,index,index);
    combination(index) = uint8(255*A(index));
    hmap = geoshow(combination,R);
    uistack(borders,'top')
    title({wmst.LayerName,time})
    frames(k) = getframe(hfig);
    hour = hour + 1/24;
end
```

Set the `Visible` property of the figure to `'on'`. View the movie loop three times at 1.5 frames per second.

```
hfig.Visible = 'on';
shg
movie(hfig,frames,3,1.5)
```



See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Add Custom Basemaps from WMS Data

This topic shows how to add custom basemaps from data stored in WMS servers by using the `addCustomBasemap` function. You can use custom basemaps with map display objects that have a `Basemap` property, including geographic axes objects and geographic globe objects. When you add a custom basemap by using the `addCustomBasemap` function, the basemap remains available for use in future MATLAB® sessions.

Add Basemap from WMS Layer

Search the WMS Database for layers from the GEBCO WMS server. Refine the list to include the `"GEBCO_LATEST_2_sub_ice_topo"` layer, which contains color-shaded elevation and under-ice topography for the globe. Synchronize the layer with the WMS server.

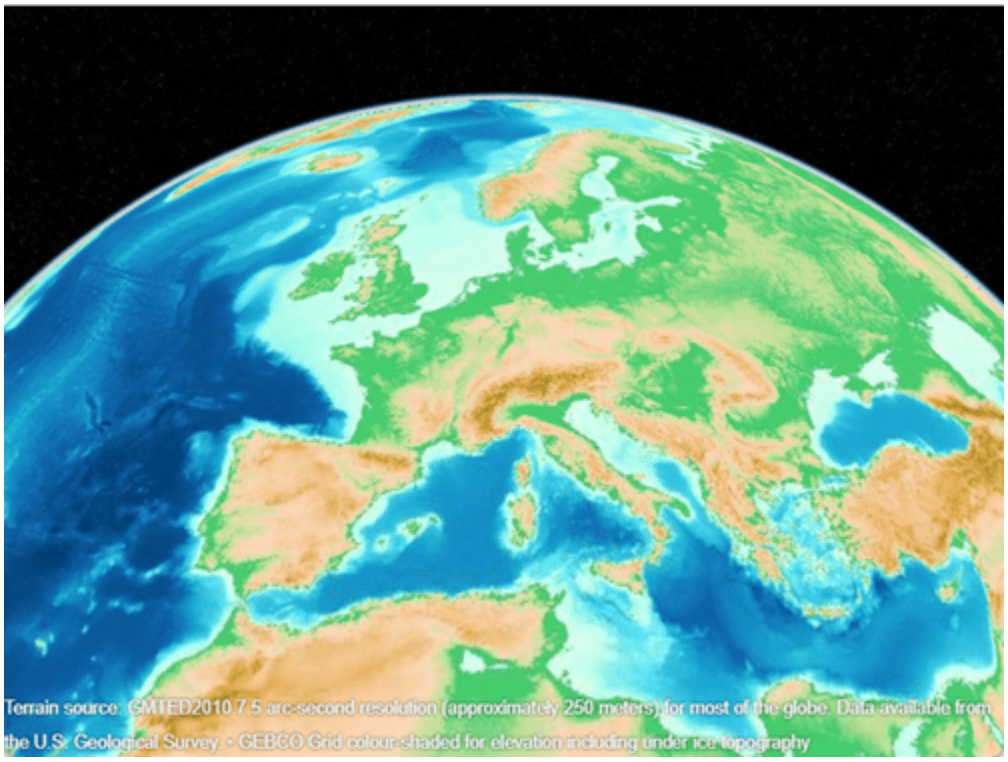
```
layers = wmsfind("gebco.net",SearchField="serverurl");
iceTopoLayer = refine(layers,"GEBCO_LATEST_2_sub_ice_topo");
iceTopoLayer = wmsupdate(iceTopoLayer);
```

Add a custom basemap from the WMS layer by using the `addCustomBasemap` function. Specify an attribution by using the `Attribution` name-value argument and the title of the layer.

```
attribution = iceTopoLayer.LayerTitle;
addCustomBasemap("gebcoIceTopo",iceTopoLayer,Attribution=attribution)
```

Create a geographic globe that uses the custom basemap. Position the camera above Europe.

```
uif = uifigure;
g = geoglobe(uif,Basemap="gebcoIceTopo");

campos(g,33,8,4e6)
campitch(g,-70)
```



Add Basemap from WMS GetMap Request URL

Search the WMS Database for layers from the NASA Earth Observations (NEO) WMS server. Refine the list to include the `"MOD_LSTD_M"` layer, which contains daytime land surface temperatures. Synchronize the layer with the WMS server.

```
layers = wmsfind("neo.gsfc.nasa.gov",SearchFields="ServerURL");
tempLayers = refine(layers,"land surface temperature");
tempDayLayer = refine(tempLayers,"MOD_LSTD_M");
tempDayLayer = wmsupdate(tempDayLayer);
```

Create a WMS map request from the layer. Request data from June 1, 2024 by updating the `Time` property of the map request.

```
mapRequest = WMSMapRequest(tempDayLayer);
mapRequest.Time = "2024-06-01";
```

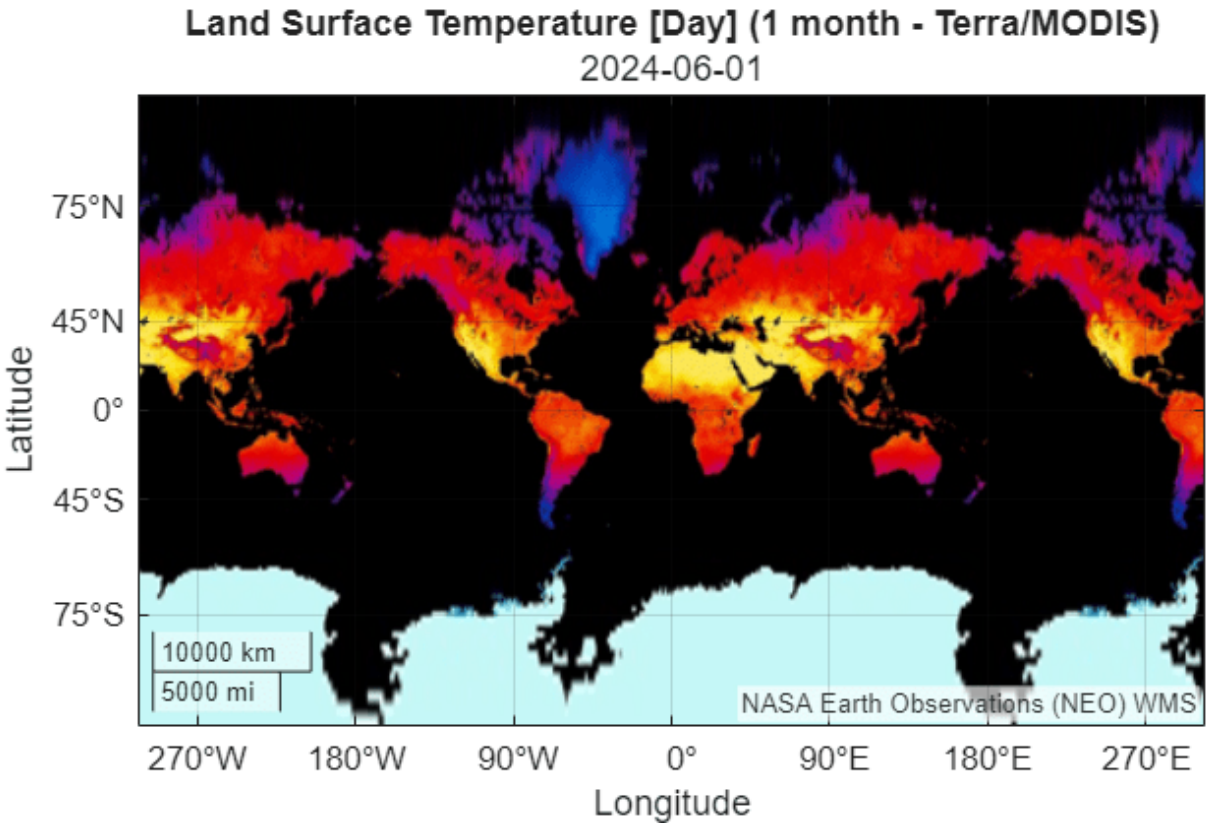
Get the WMS GetMap request URL from the map request. Then, add a custom basemap from the URL by using the `addCustomBasemap` function. Specify an attribution by using the `Attribution` name-value argument and the title of the server.

```
url2024 = mapRequest.RequestURL;
attribution = tempDayLayer.ServerTitle;
addCustomBasemap("landTempJune2024",url2024,Attribution=attribution)
```

Display the basemap in a geographic axes object by using the `geobasemap` function. Add a title and subtitle using the title of the layer and the time stored in the map request, respectively.

```
figure
geobasemap landTempJune2024
```

```
t = tempDayLayer.LayerTitle;
title(t)
subtitle(mapRequest.Time)
```

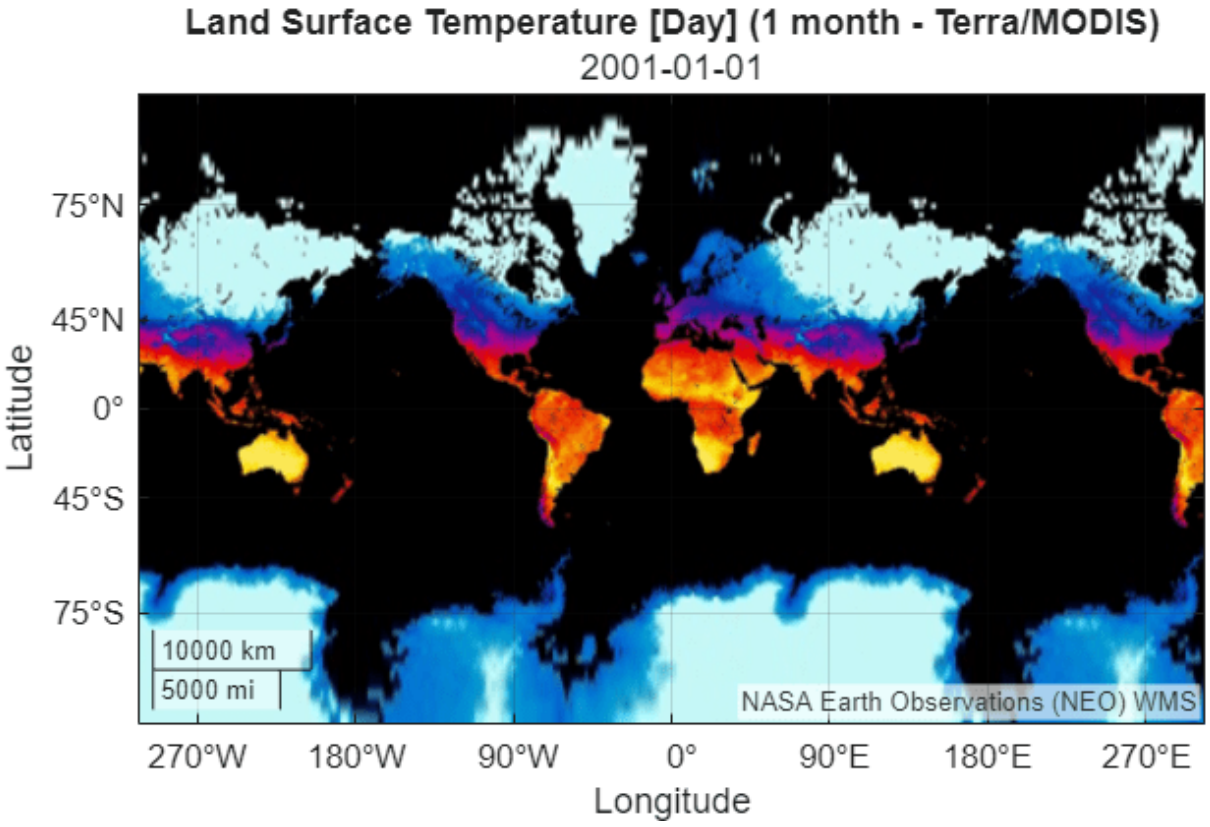


Add a second basemap using temperatures from January 1, 2001. Update the `Time` property of the map request, get the updated WMS GetMap request URL, and add the basemap using the updated URL.

```
mapRequest.Time = "2001-01-01";
url2000 = mapRequest.RequestURL;
addCustomBasemap("temperatureJan2001",url2000,Attribution=attribution)
```

Display the second basemap in a new geographic axes object.

```
figure
geobasemap temperatureJan2001
title(t)
subtitle(mapRequest.Time)
```



See Also

Functions
`addCustomBasemap` | `wmsfind` | `wmsupdate` | `geoglobe` | `geobasemap`

Objects
`WMSLayer` | `WMSMapRequest` | `GeographicGlobe`

Properties
GeographicAxes Properties

Related Examples

- “Basic Workflow for Creating WMS Maps” on page 7-3

Modify Your WMS Map Request

In this section...
“Set Map Request Geographic Limits and Time” on page 7-36 “Manually Edit Web Map Request URL” on page 7-37

Set Map Request Geographic Limits and Time

A `WMSMapRequest` object contains properties to modify the geographic extent and time of the requested map. This example shows how to modify your map request to map land surface temperature for the southern tip of Africa. For a complete list of properties, see `WMSMapRequest`.

Search the WMS Database for layers on the NASA Earth Observations (NEO) WMS server. Refine the search to include only layers with `"land surface temperature"` in the layer title or layer name. Refine the search again to include only the layer with monthly values from the MODIS sensor on the Terra satellite.

```
neo = wmsfind("neo*nasa*wms",SearchFields="serverurl");
landTemp = refine(neo,"land surface temperature");
landTemp = refine(landTemp,"night*month*modis");
landTemp = wmsupdate(landTemp);
```

Create a `WebMapServer` object from the server URL stored in the `ServerURL` property of the layer.

```
server = WebMapServer(landTemp.ServerURL);
```

Create a `WebMapRequest` object from the layer and web map server object. Set the latitude and longitude limits by using the `Latlim` and `Lonlim` properties. Set the time request to March 1, 2009 by using the `Time` property.

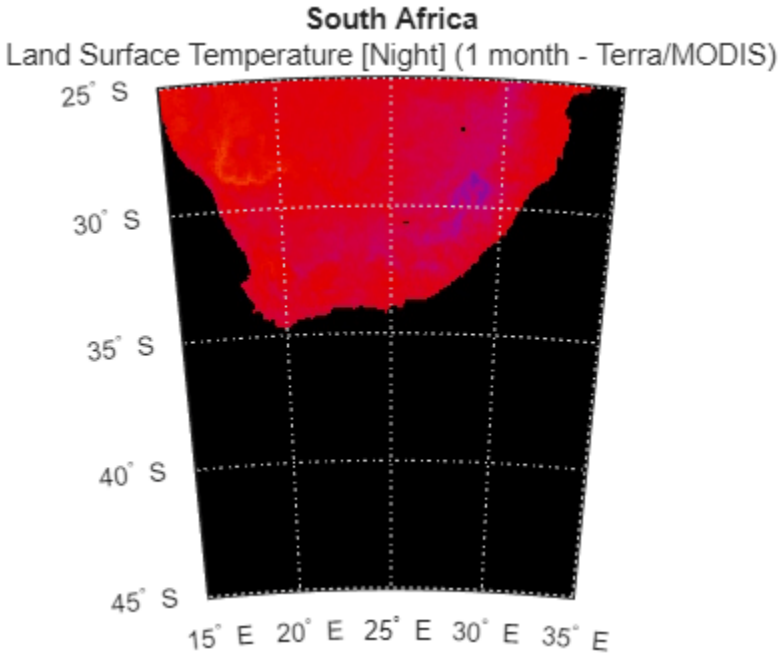
```
mapRequest = WMSMapRequest(landTemp,server);
mapRequest.Latlim = [-45 -25];
mapRequest.Lonlim = [15 35];
mapRequest.Time = "2009-03-01";
```

Send your request to the server by using the `getMap` function.

```
landTempImage = getMap(server,mapRequest.RequestURL);
```

Display the image on a map.

```
figure
worldmap(mapRequest.Latlim,mapRequest.Lonlim);
setm(gca,"mlabelparallel",-45)
geoshow(landTempImage,mapRequest.RasterReference);
title("South Africa",landTemp.LayerTitle,Interpreter="none")
```



Read abstract information for the layer from the `MetadataURL` field.

```
readerFcn = @(filename)readstruct(filename,FileType="xml");
options = weboptions(ContentReader=readerFcn,Timeout=10);
xml = webread(landTemp.Details.MetadataURL,options);
abstract = xml.idinfo.descript.abstract;
```

Manually Edit Web Map Request URL

You can modify a map request URL manually.

Search the WMS Database for a layer containing terrain elevation data from the WMS server hosted by MathWorks®. Get the map request URL for the layer.

```
layers = wmsfind("mathworks","SearchField","serverurl");
layer = refine(layers,"elevation");
mapRequest = WMSMapRequest(layer);
```

Set the map request URL to a variable.

```
mapURL = mapRequest.RequestURL;
```

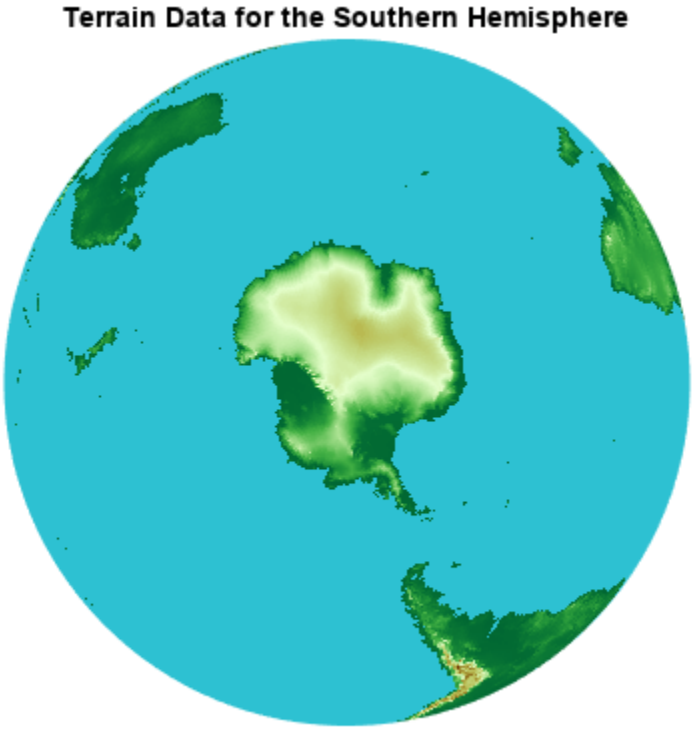
Specify a background color and request data for the southern hemisphere by manually editing the map request URL. To do this, copy and paste the contents of `mapURL` into a new variable. Then, change the background color section of the URL to `&BGCOLOR=0x2DC1D2` and change the bounding box section of the URL to `&BBOX=-90.0,-180,0,180.0`.

```
modifiedURL = ['https://wms.mathworks.com?SERVICE=WMS' ...
                '&LAYERS=terrain&CRS=EPSG:4326&FORMAT=image/png' ...
                '&TRANSPARENT=FALSE&HEIGHT=256&BGCOLOR=0x2DC1D2' ...
```

```
                '&REQUEST=GetMap&WIDTH=512&BBOX=-90.0,-180,0,180.0' ...
                '&STYLES=&VERSION=1.3.0'];
```

Read and display the modified map.

```
[A,R] = wmsread(modifiedURL);
figure
axesm globe
axis off
geoshow(A,R)
title("Terrain Data for the Southern Hemisphere")
```



See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Read WMS Maps Using Different Coordinate Reference Systems

To read a WMS map in the EPSG:4326 coordinate reference system, use the `wmsread` function. EPSG:4326 is based on the 1984 World Geodetic System (WGS84) datum. All servers in the WMS Database, and presumably all WMS servers in general, use the EPSG:4326 coordinate reference system. This system is a requirement of the OGC® WMS specification.

To read a WMS map in a different coordinate reference system, create a request URL using `WebMapServer` and `WMSMapRequest` objects and then read the map using the `getMap` method. For more information about coordinate reference systems, see Spatial Reference.

This example shows how to read a WMS map with data in Web Mercator coordinates, also known as WGS 84/Pseudo-Mercator coordinates. The Web Mercator coordinate system is commonly used by web applications.

Find Web Mercator Coordinates for Region

Import a GeoTIFF image of Boston as an array and a `MapCellsReference` object. Find the projected coordinate reference system for the image.

```
[A,R] = readgeoraster("boston.tif");
p_geotiff = R.ProjectedCRS;
```

Unproject the *x*- and *y*-world limits and then reproject them to use the Web Mercator coordinate reference system (EPSG:3857).

```
[latlim,lonlim] = projinv(p_geotiff,R.XWorldLimits,R.YWorldLimits);
p_webmercator = projcrs(3857);
[xlimits,ylimits] = projfwd(p_webmercator,latlim,lonlim);
```

To obtain imagery in this coordinate reference system, you must use `WMSMapRequest` and `WebMapServer` objects because the `wmsread` function only reads data in the WGS84 coordinate reference system (EPSG:4326).

Read and Display Map for Region

The USGS National Map provides orthoimagery and topography maps for various regions of the United States. The USGS Imagery Only server provides data in both WGS84 coordinates and Web Mercator coordinates.

Search the WMS Database for the USGS Imagery Only server and select the first layer.

```
doqLayer = wmsfind("usgsimageryonly",SearchField="serverurl");
doqLayer = wmsupdate(doqLayer);
```

Create `WebMapServer` and `WMSMapRequest` objects.

```
server = WebMapServer(doqLayer.ServerURL);
request = WMSMapRequest(doqLayer,server);
```

Modify the map request by setting properties of the `WMSMapRequest` object. For this example, specify an image height and width for a sample size of 5 meters. Set the map limits to cover the same region as the GeoTIFF file.

```
metersPerSample = 5;
h = round(diff(ylimits)/metersPerSample);
w = round(diff(xlimits)/metersPerSample);

ylimits = [ylimits(1), ylimits(1) + h*metersPerSample];
xlimits = [xlimits(1), xlimits(1) + w*metersPerSample];

request.CoordRefSysCode = "EPSG:3857";
request.ImageHeight = h;
request.ImageWidth  = w;
request.XLim = xlimits;
request.YLim = ylimits;
```

Read a map of the orthoimagery in Web Mercator coordinates.

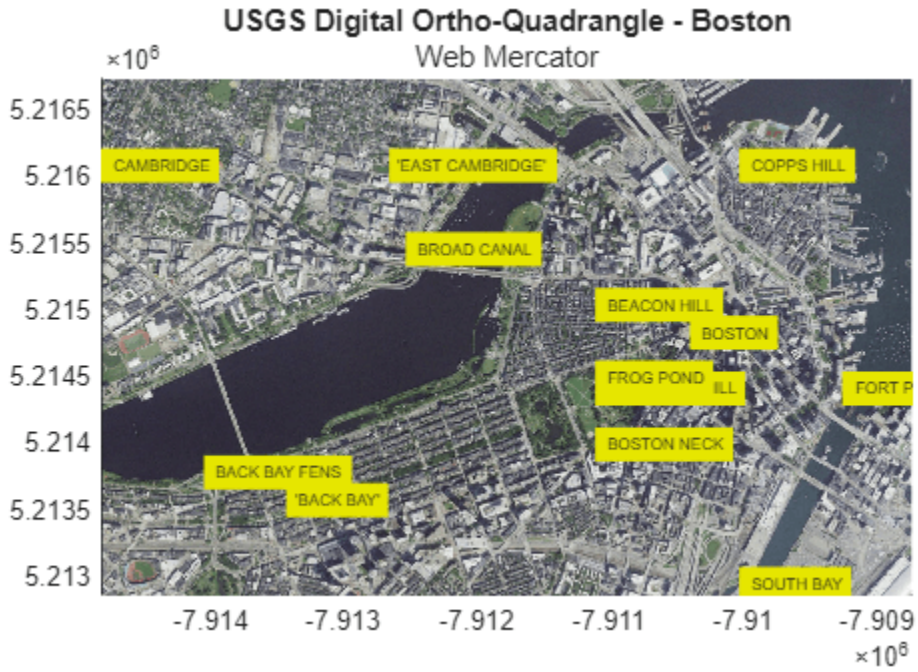
```
A_webmercator = getMap(server,request.RequestURL);
R_webmercator = request.RasterReference;
```

Display the orthoimagery on a map.

```
figure
mapshow(A_webmercator,R_webmercator)
axis tight
title("USGS Digital Ortho-Quadrangle - Boston","Web Mercator")
```

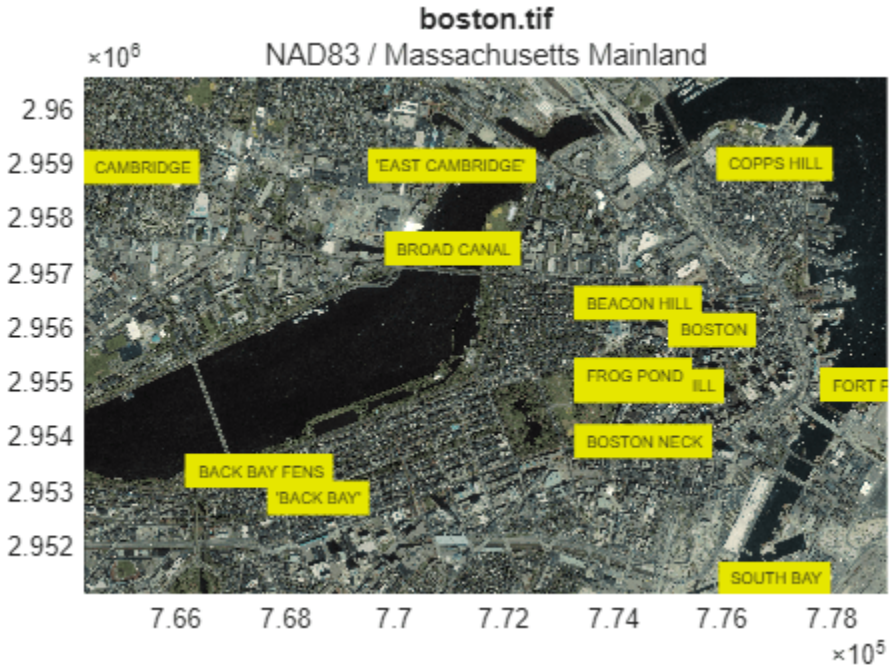
Read Boston place names from a shapefile. Unproject and reproject the place names so that they are in Web Mercator coordinates. Display the place names on the map.

```
S = shaperead("boston_placenames.shp");
x_names = [S.X] * unitsratio("sf","meter");
y_names = [S.Y] * unitsratio("sf","meter");
names = {S.NAME};

[lat_placenames,lon_placenames] = projinv(p_geotiff, ...
                                          x_names,y_names);
[x_webmercator,y_webmercator] = projfwd(p_webmercator, ...
                                        lat_placenames,lon_placenames);

text(x_webmercator,y_webmercator,names, ...
    BackgroundColor=[0.9 0.9 0],FontSize=6,Clipping="on")
```



Compare the map you read from the server to the GeoTIFF image.

```
figure
mapshow("boston.tif")
axis tight
title("boston.tif",p_geotiff.Name)
text(x_names,y_names,names, ...
    BackgroundColor=[0.9 0.9 0],FontSize=6,Clipping="on")
```

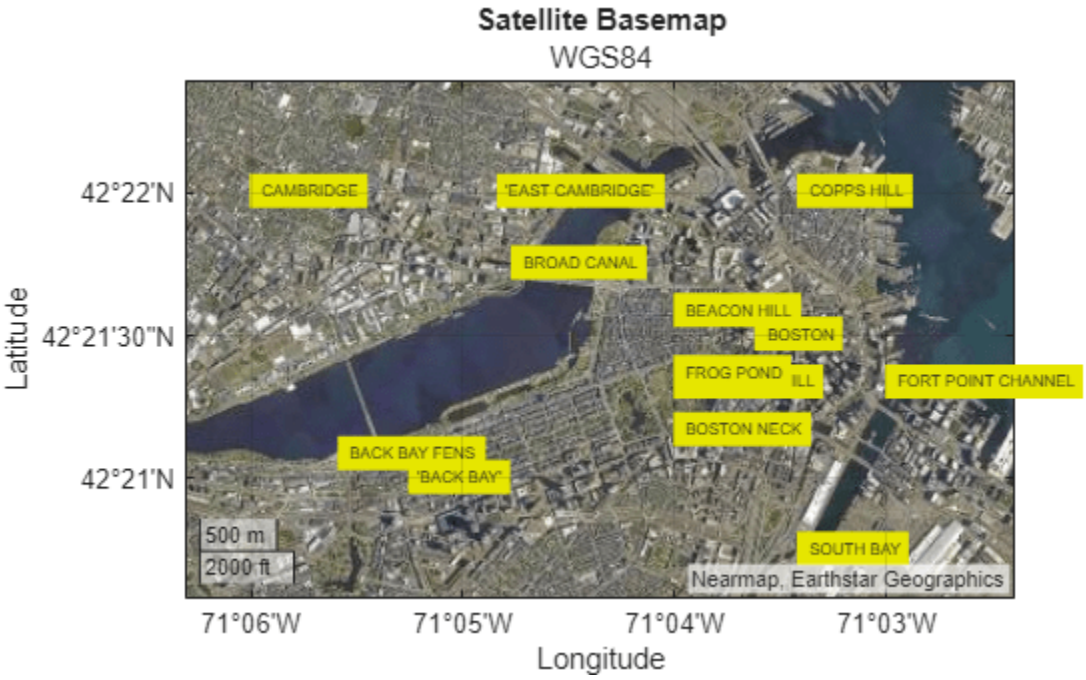


You can also compare the maps by using geographic axes and a satellite basemap. When you plot on geographic axes, coordinates are referenced to the WGS84 coordinate reference system.

```
figure
geolimits(latlim,lonlim)
geobasemap satellite
text(lat_placenames,lon_placenames,names, ...
    BackgroundColor=[0.9 0.9 0],FontSize=6)
title("Satellite Basemap","WGS84")
```

View all of the text labels by changing the geographic limits.

```
geolimits([42.3430 42.3732],[-71.1052 -71.0399])
```



See Also

Functions
`wmsfind` | `wmsupdate` | `wmsread` | `getMap`

Objects
`WMSMapRequest` | `WebMapServer` | `WMSLayer`

Related Examples

- “Modify Your WMS Map Request” on page 7-36

Read Quantitative Data from WMS Server

WMS servers typically provide layers as images instead of as quantitative data. In some rare cases, you can get quantitative data from specific WMS servers by requesting data in the GeoTIFF or BIL format.

You can determine the formats that a server supports by querying the `ImageFormats` field of the structure contained within the `Details` property of a WMS layer.

```
layers = wmsfind("mathworks","SearchField","serverurl");
elevation = refine(layers,"elevation");
elevation = wmsupdate(elevation);
elevation.Details.ImageFormats
```

These examples show how to read terrain elevation data from the WMS server hosted by MathWorks and then display the data as a surface or a texture-mapped surface.

Read and Display Terrain Elevation Data

Search the WMS Database for a layer containing terrain elevation data from the WMS server hosted by MathWorks.

```
layers = wmsfind("mathworks",SearchField="serverurl");
elevation = refine(layers,"elevation");
```

Read the terrain elevation data from the server for a region surrounding the Gulf of Maine. Specify the sampling interval as `30` arc-seconds and the image format as BIL.

```
GT = geocode("Gulf of Maine","physical",FilterResults="first");
[latlim,lonlim] = bounds(GT.Shape);

samplingInterval = dms2degrees([0 0 30]);
imageFormat = "image/bil";

[Z,R] = wmsread(elevation,Latlim=latlim,Lonlim=lonlim, ...
    ImageFormat=imageFormat,CellSize=samplingInterval);
```

Note that `Z` is an array containing terrain elevation data rather than an RGB image. Find the minimum and maximum elevation.

```
elevationLimits = [min(min(Z)) max(max(Z))]
```

```
elevationLimits = 1×2 int16 row vector

      -7    1181
```

Prepare the data for plotting by converting the data type to `double`. To make the water areas appear blue on the plot, set elevations at sea level (`Z == 0`) to a value below sea level.

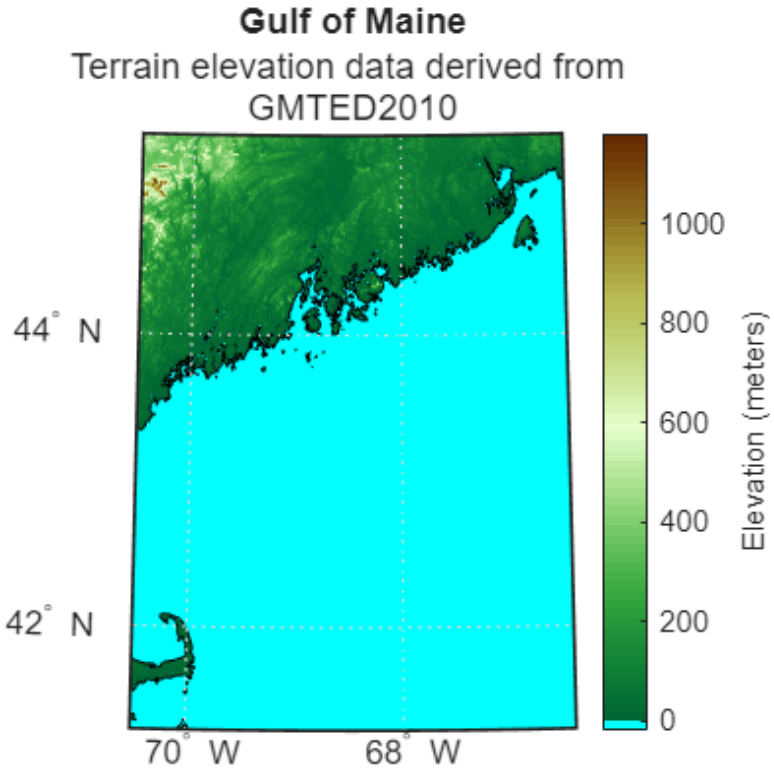
```
Z = double(Z);
Z(Z == 0) = -1;
```

Display the terrain elevation data on a map and plot contour lines at sea level.

```
figure
worldmap(Z,R)
geoshow(Z,R,DisplayType="texturemap")
contourm(Z,R,[0 0],Color="k")
```

Customize the map by adding a title and subtitle, a colormap appropriate for elevation data, and a labeled color bar.

```
title("Gulf of Maine")
st = textwrap(string(elevation.LayerTitle),40);
subtitle(st)
demcmap(Z)

c = colorbar;
c.Label.String = "Elevation (meters)";
```



Merge Elevation Data with Rasterized Vector Data

Read terrain elevation data and rasterized national boundaries for a region. Then, embed the boundaries into the elevation data and display the result on a map.

Read Terrain Elevation Data

Search the WMS Database for a layer containing terrain elevation data from the WMS server hosted by MathWorks.

```
layers = wmsfind("mathworks",SearchField="serverurl");
elevation = refine(layers,"elevation");
```

Read the terrain elevation data from the server for a region surrounding Afghanistan. Specify the sampling interval as 1 minute and the image format as BIL.

```
latlim = [25 40];
lonlim = [55 80];
samplingInterval = dms2degrees([0 1 0]);
[ZA,R] = wmsread(elevation,Latlim=latlim,Lonlim=lonlim, ...
   CellSize=samplingInterval,ImageFormat="image/bil");
```

Note that `ZA` is an array of elevation data rather than an RGB image. Find the minimum and maximum elevations.

```
elevationLimits = [min(min(ZA)) max(max(ZA))]
```

```
elevationLimits = 1×2 int16 row vector

     -13    7350
```

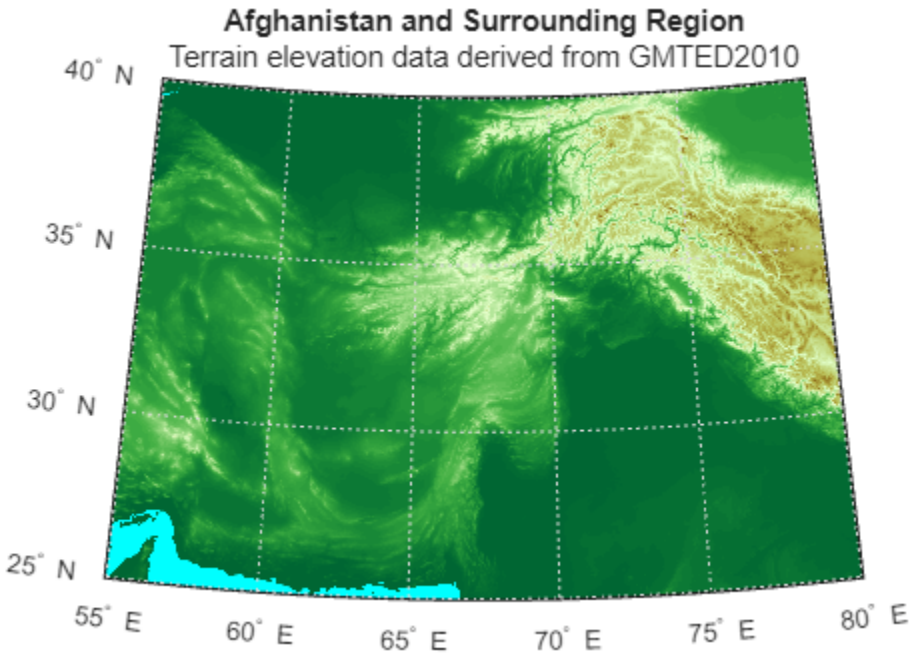
Display Terrain Elevation Data

Prepare the data for plotting by converting the data type to `double`. To make the water areas appear blue on the plot, set elevations at sea level (`ZA == 0`) to a value below sea level.

```
ZA = double(ZA);
ZA(ZA == 0) = -1;
```

Display the elevation data as a texture map.

```
figure
worldmap("Afghanistan")
geoshow(ZA,R,DisplayType="texturemap")
demcmap(ZA)
title("Afghanistan and Surrounding Region")
subtitle(elevation.LayerTitle)
```



Read Rasterized Vector Data

Get a layer containing boundaries data from the EOC Basemap Map Service server.

```
info = wmsinfo("https://geoservice.dlr.de/eoc/basemap/wms?");
layers = info.Layer;
boundaries = refine(layers,"ne_countries");
```

Read the boundaries as an RGB image. Dark values in the image represent the boundaries.

```
B = wmsread(boundaries,Latlim=latlim,Lonlim=lonlim,CellSize=samplingInterval);
```

Merge Elevation and Rasterized Vector Data

To embed the boundaries into the elevation data, find the elements of the boundaries image that contain dark values. Then, replace the corresponding elements of the elevation array with a value slightly higher than the maximum elevation.

```
ZB = ZA;
idx = B(:,:,1) < 250;
ZB(idx) = max(ZA(:)) + 1;
```

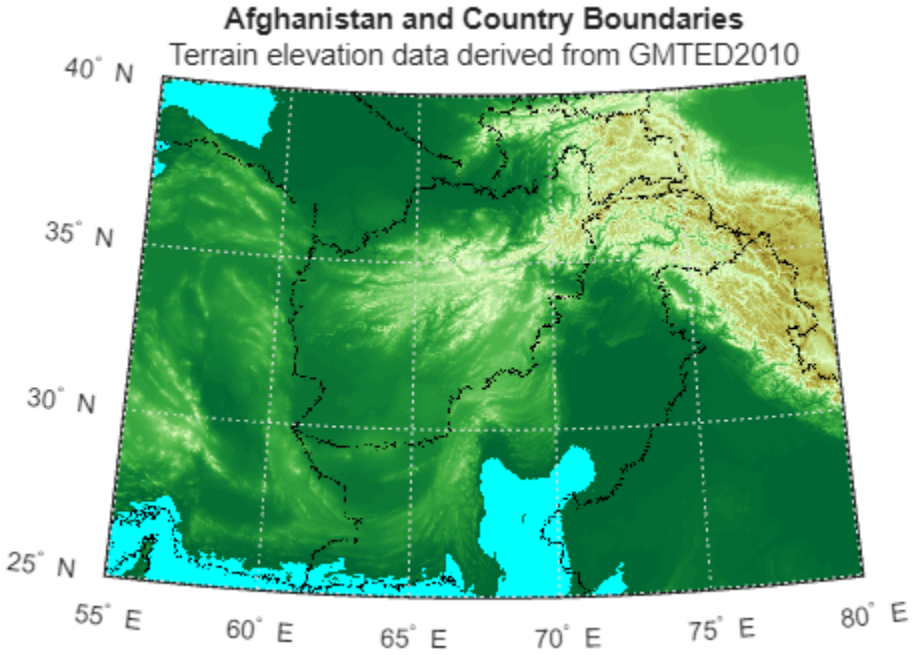
Display Merged Data

Display the updated data on a new map.

```
figure
worldmap("Afghanistan")
geoshow(ZB,R,DisplayType="texturemap")
title("Afghanistan and Country Boundaries")
subtitle(elevation.LayerTitle)
```

Apply a colormap appropriate for elevation data. Use black for the boundaries.

```
cmap = demcmap(ZB);
cmap(end,:,:) = [0 0 0];
colormap(cmap)
```



Drape Basemap Imagery onto Elevation Data

Read basemap imagery and terrain elevation data for a region. Then, create a 3-D map display by draping the basemap imagery over the elevation data.

Read Basemap Imagery

Specify the latitude and longitude limits for a region around the Grand Canyon. Read an image for the region from the `"satellite"` basemap as an array and as a map cells reference object in Web Mercator coordinates.

```
latlim = [36 36.23];
lonlim = [-113.36 -113.13];
[A,R,attrib] = readBasemapImage("satellite",latlim,lonlim);
```

Read Terrain Elevation Data

WMS servers typically provide data referenced to the World Geodetic System of 1984 (WGS84). If a layer provides data referenced to a different coordinate system, then you can specify the coordinate system by using a web map server object, a WMS map request object, and the `getMap` function.

Create a web map server object that communicates with the WMS server provided by MathWorks. Get the terrain elevation layer from the capabilities document.

```
serverURL = "https://wms.mathworks.com";
server = WebMapServer(serverURL);
```

```
info = getCapabilities(server);
elevation = refine(info.Layer,"elevation");
```

Verify the server provides data referenced to Web Mercator coordinates. The Web Mercator coordinate system has the EPSG code 3857.

```
elevation.CoordRefSysCodes
```

```
ans = 2×1 cell
    {'EPSG:3857'}
    {'EPSG:4326'}
```

Create a map request object for the layer.

- Specify the coordinate system as Web Mercator.
- Specify the image dimensions and world *xy*-limits so that they match the basemap image.
- Request quantitative elevation data by specifying the image format as BIL.

```
mapRequest = WMSMapRequest(elevation);
mapRequest.CoordRefSysCode = "EPSG:3857";
mapRequest.ImageHeight = size(A,1);
mapRequest.ImageWidth = size(A,2);
mapRequest.XLim = R.XWorldLimits;
mapRequest.YLim = R.YWorldLimits;
mapRequest.ImageFormat = "image/bil";
```

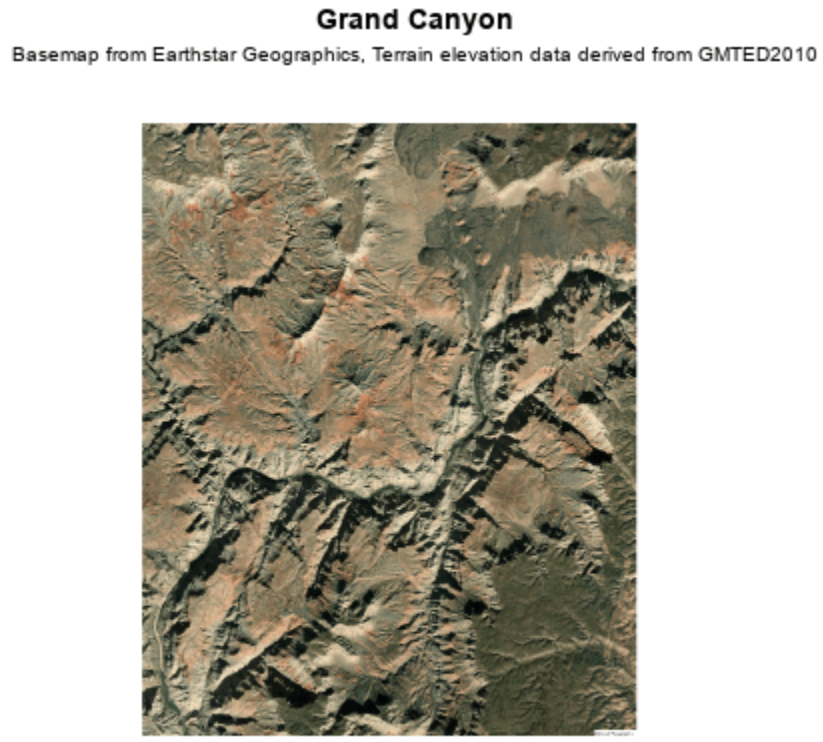
Read the elevation data by using the `getMap` function.

```
Z = getMap(server,mapRequest.RequestURL);
Z = double(Z);
```

Display Imagery and Elevation Data

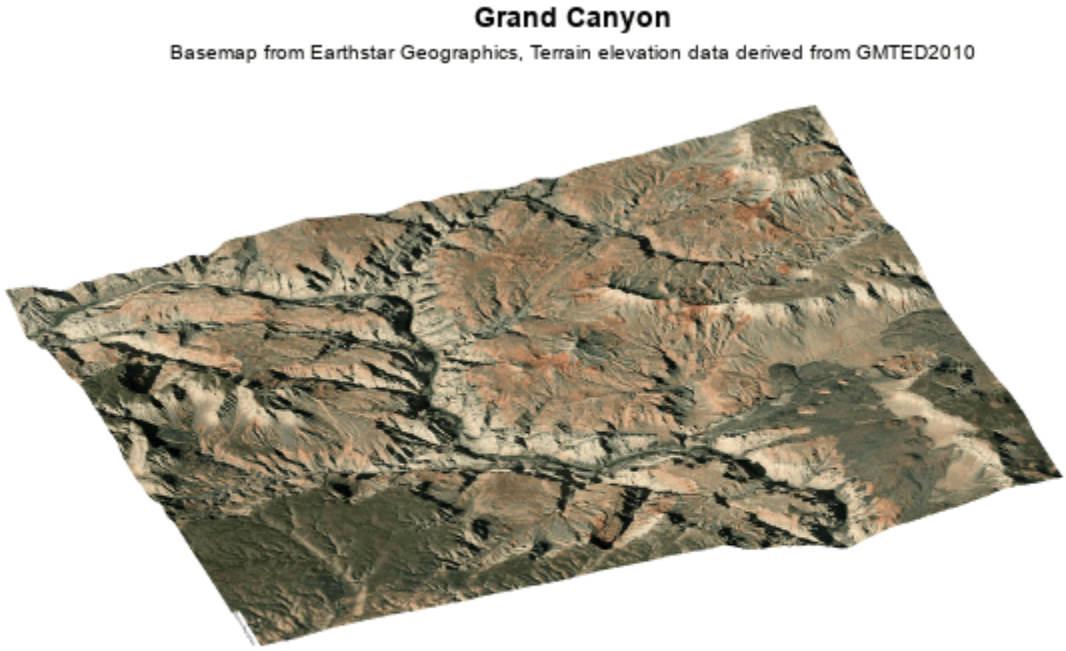
Drape the image over the elevation data.

```
figure
mapshow(Z,R,"DisplayType","surface","CData",A)
bmapAttrib = "Basemap from " + attrib;
wmsAttrib = string(elevation.LayerTitle);
title("Grand Canyon")
subtitle(bmapAttrib + ", " + wmsAttrib,"FontSize",8)
axis off
```



Adjust the camera view.

```
campos([-12400000 4240000 152000])
camva(5.4)
```



Tips

- To request quantitative data, you must generally create either a Web Coverage Service (WCS) request or a Web Feature Service (WFS) request. Mapping Toolbox does not support WCS or WFS requests.
- Although BIL files can contain multiple co-registered bands (channels), the BIL files returned by WMS servers include only a single band.
- The `wmsread` function interpolates the elevation data from the WMS server hosted by MathWorks based on the specified latitude and longitude limits. As a result, the data associated with a geographic location varies slightly depending on the limits used to read the data.
- The elevation data from the WMS server hosted by MathWorks does not contain bathymetry data.
- The elevation data from the WMS server hosted by MathWorks does not contain data for a small region within Lake Superior.
- Map request URLs from the WMS server hosted by MathWorks are not supported by web browsers or by the `webread` function.

See Also

Functions
`wmsfind` | `wmsread` | `getMap`

Objects
`WebMapServer` | `WMSMapRequest`

Related Examples

- “Basic Workflow for Creating WMS Maps” on page 7-3

Write WMS Layers to KML File

Some WMS server implementations, such as GeoServer, can render their maps in a non-image format, such as KML. KML is an XML dialect used by Google Earth™ and Google Maps™ browsers. The `WebMapServer.getMap` method and the `wmsread` function do not allow you to use the KML format because they import only standard graphics image formats. Work around this limitation by using the `WMSMapRequest.RequestURL` property.

Get the capabilities document for the MassGIS WMS server. Get the list of layers from the capabilities document. Then, refine the layers to find a FEMA Flood Zone layer.

```
geoserver = wmsinfo("https://gis-prod.digital.mass.gov/geoserver/wms?");
massgis = geoserver.Layer;
floodzone = refine(massgis,"FEMA Flood Zones",SearchField="LayerTitle");
floodzone = floodzone(1);
```

Specify geographic limits for a region around Boston, Massachusetts.

```
latlim = [ 42.305  42.417];
lonlim = [-71.131 -70.99];
```

Create a `WMSMapRequest` object and set the geographic limits.

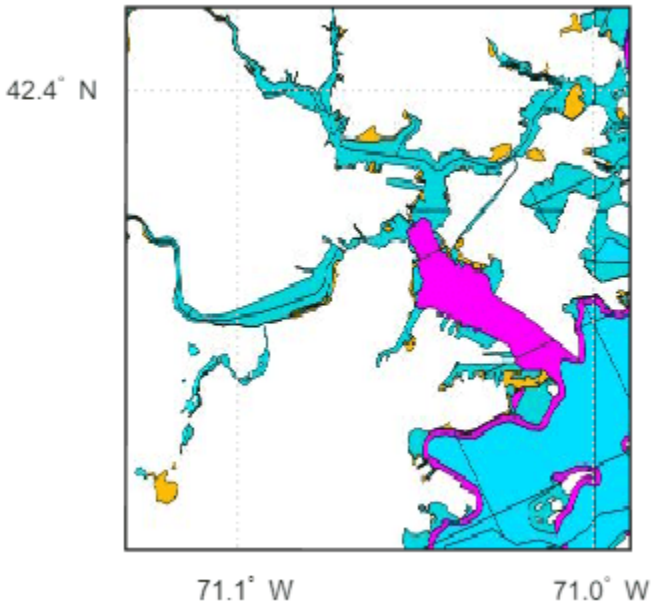
```
request = WMSMapRequest(floodzone);
request.Latlim = latlim;
request.Lonlim = lonlim;
```

Read the graphics image from the server.

```
[A,R] = wmsread(request.RequestURL);
```

Display the image on a map.

```
figure
usamap(A,R)
geoshow(A,R)
```



Request an image format that opens in Google Earth.

```
request.ImageFormat = "application/vnd.google-earth.kml+xml";
```

Use the `websave` function to write out a KML file.

```
filename = "floodzone.kml";
websave(filename,request.RequestURL);
```

Open the file with Google Earth to view. On Windows® platforms, you can display the KML file by uncommenting this code.

```
% winopen(filename)
```

For UNIX® and Mac users, you can display the KML file by uncommenting this code.

```
% cmd = 'googleearth ';
% fullfilename = fullfile(pwd, filename);
% system([cmd fullfilename])
```

See Also

`wmsfind` | `wmsupdate` | `wmsread` | `websave`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Troubleshoot WMS Servers

In this section...

“Connection Errors” on page 7-54
“Intermittent Internet Access” on page 7-55
“Geographic Limits in Descending Order or Out of Bounds” on page 7-55
“Non-EPSG:4326 Coordinate Reference Systems” on page 7-55
“Map Not Returned” on page 7-56
“Unsupported WMS Version” on page 7-57
“Other Unrecoverable Server Errors” on page 7-57

Connection Errors

One of the challenges of working with WMS is that sometimes you can have trouble connecting to a server.

Time-Out Error

A server might issue a time-out error such as `Connection timed out: connect` or `Read timed out`.

Workaround: Try setting the `'TimeoutInSeconds'` parameter of the `wmsread`, `wmsinfo`, or `wmsupdate` function to a larger value. The time-out setting defaults to 60 seconds.

Unavailable Server

A server might be temporarily unavailable or busy, or the WMS server application might experience an issue. In some cases, the server issues an HTTP response code of 500, such as:

```
Server returned HTTP response code: 500 for URL: http://xyz.com ...
```

Workarounds:

- Try again later.
- Try requesting a different image format by setting the `'ImageFormat'` parameter.
- Try connecting multiple times by using a `while` loop and `try/catch` block. For example, this code makes five attempts to get information about the USGS National Map Seamless server. If the connection fails after five attempts, then the code errors.

```
numberOfAttempts = 5;
attempt = 0;
info = [];
serverURL = "http://basemap.nationalmap.gov/ArcGIS/services/" + ...
    "USGSImageryOnly/MapServer/WMSServer?";
while(isempty(info))
    try
        info = wmsinfo(serverURL);
        orthoLayer = info.Layer(1);
    catch e

        attempt = attempt + 1;
```

```
            if attempt > numberOfAttempts
                throw(e);
            else
                fprintf('Attempting to connect to server:\n"%s"\n', serverURL)
            end
        end
    end
```

Intermittent Internet Access

You have intermittent Internet access or you want to share WMS data with someone without Internet access.

Workaround: Reading data from WMS servers requires Internet access. However, once you read data from a server, you can save the data as a MAT or GeoTIFF file. Once you save the data, you can load it without reading it again from the server. For more information, see “Create WMS Maps When Internet Access Is Intermittent” on page 7-6.

Geographic Limits in Descending Order or Out of Bounds

Some servers do not follow the OGC specification guidelines regarding latitude and longitude limits.

The OGC specification requires, and the WMS functions expect, that the limits are ascending. However, some sites have descending limits. As a result, you might see an error such as:

```
"??? Error using ==> WMSMapRequest>validateLimit at 1313
Expected the elements of 'Latlim' to be in ascending order."
```

Additionally, some servers have limits that exceed the bounds of [-180,180] for longitude and [-90,90] for latitude.

Workaround: Set the `Latlim` and `Lonlim` properties of the `WMSLayer` object:

```
layers = wmsfind('bluemarbleng');
layer = wmsupdate(layers(1));
latlim = [min(layer.Latlim), max(layer.Latlim)];
lonlim = [min(layer.Lonlim), max(layer.Lonlim)];
layer.Latlim = [max([ -90, latlim(1)]), min([ 90, latlim(2)])];
layer.Lonlim = [max([-180, lonlim(1)]), min([180, lonlim(2)])];
[A,R] = wmsread(layer);
```

You must update your layer before setting the limits. Otherwise, the `wmsread` function updates the limits from the server, and you once again have unsupported limits.

Non-EPSG:4326 Coordinate Reference Systems

Some layers are not defined in the EPSG:4326 or CRS:84 coordinate reference system. Reading these layers with the `wmsread` function is not supported.

Workaround: Construct a request URL by using the `WMSMapRequest` object and read the layer by using the `getMap` object function. For more information, see “Read WMS Maps Using Different Coordinate Reference Systems” on page 7-39.

Map Not Returned

Sometimes you can connect to the WMS server, but you do not receive the map you are expecting.

Blank Map Returned

A server might return a blank map.

Workaround: You can change the scale of your map; either increase the image height and width or change the geographic bounds. Another possibility is that your requested geographic extent lies outside the extent of the layer, in which case you should change the extent of your request. A third possibility is that you have the wrong image format selected; in this case, change the `'ImageFormat'` parameter.

HTML File Returned

You might see this error message:

```
The server returned an HTML file instead of an image file.
```

Workaround: Follow the directions in the error message. The following example, which uses a sample URL, illustrates the type of error message you receive.

```
% Example command.
[A,R] = wmsread(['https://www.mathworks.com?',...
'&BBOX=-180,-90,180,90&CRS=EPSG:4326&VERSION=1.1.1']);
```

Sample error message:

```
Error using WebMapServer>issueReadGetMapError (line 974)
The server returned an HTML file instead of an image file.
You may view the complete error message by issuing the command,
 web('https://www.mathworks.com?&BBOX=-180,-90,180,90&CRS=EPSG:4326&VERSION=1.1.1')
  or
 urlread('https://www.mathworks.com?&BBOX=-180,-90,180,90&CRS=EPSG:4326&VERSION=1.1.1').

Error in WebMapServer>readImageFormat (line 874)
        issueReadGetMapError(filename, requestURL);

Error in WebMapServer>readGetMapFile (line 852)
    A = readImageFormat(filename, requestURL);

Error in WebMapServer/getMap (line 299)
            A = readGetMapFile(filename, h.RequestURL);

Error in wmsread (line 376)
A = server.getMap(mapRequestURL);
```

XML File Returned

The server issues a very long error message, beginning with this phrase:

```
An error occurred while attempting to get the map from the server.
The error returned is <?xml version="1.0" encoding="utf-8"?> ...
```

Workaround: This problem occurs because the server breaks with the requirements of the OGC standard and returns the XML capabilities document rather than the requested map. Choose a different layer or server.

Unsupported WMS Version

In rare cases, the server uses a different and unsupported WMS version. In this case, you receive an error message such as:

```
The WMS version, '1.2.0', listed in layer.Details.Version is not
supported by the server. The supported versions are: '1.0.0' '1.1.0'
'1.1.1' '1.3.0'.
```

Workaround: Choose a different server.

Other Unrecoverable Server Errors

The server issues an error indicating that no correction or workaround exists. These cases result in the following types of error messages:

```
Server redirected too many  times (20)

An error occurred while attempting to parse the XML capabilities
document from the server.

Unexpected end of file from server

An error occurred while attempting to get the map from the server.
The server returned a map containing no data.
```

See Also

`wmsfind` | `wmsupdate` | `wmsread`

More About

- “Basic Workflow for Creating WMS Maps” on page 7-3

Composite and Animate Web Map Service (WMS) Meteorological Layers

This example shows how to composite and animate data from multiple Web Map Service (WMS) layers.

The base layer is from the NASA Goddard Space Flight Center's Scientific Visualization Studio (SVS) Image Server. The data in this layer shows satellite cloud data during Hurricane Katrina from August 23 through August 30, 2005. The layer consists of cloud data extracted from GOES-12 imagery and overlaid on a color image of the southeast United States.

Next-Generation Radar (NEXRAD) images, collected by the Iowa State University's Iowa Environmental Mesonet (IEM) Web map server, are composited with the cloud data at regular intervals of time.

In particular, this example will show you how to:

- Use the WMS database to find the Katrina and NEXRAD layers
- Retrieve the Katrina base map from a WMS server at a particular time-step
- Retrieve the NEXRAD map from a WMS server at the same time-step
- Composite the base map with the map containing the NEXRAD imagery
- View the composited map in a projected coordinate system
- Retrieve, composite, and animate multiple time sequences
- Create a video file and animated GIF file of the animation

Understanding Basic WMS Terminology

If you are new to WMS, several key concepts are important to understand and are listed here.

- *Web Map Service* --- The Open Geospatial Consortium (OGC) defines a Web Map Service (WMS) to be an entity that "produces maps of spatially referenced data dynamically from geographic information."
- *WMS server* --- A server that follows the guidelines of the OGC to render maps and return them to clients
- *map* --- The OGC definition for map is "a portrayal of geographic information as a digital image file suitable for display on a computer screen."
- *layer* --- A dataset of a specific type of geographic information, such as temperature, elevation, weather, orthophotos, boundaries, demographics, topography, transportation, environmental measurements, and various data from satellites
- *capabilities document* --- An XML document containing metadata describing the geographic content offered by a server

Internet Access

Since WMS servers are located on the Internet, this example can be set to access the Internet to dynamically render and retrieve maps from WMS servers or it can be set to use data previously retrieved from the Internet using the WMS capabilities but now stored in local files. You can use a variable, `useInternet`, to determine whether to read data from locally stored files, or retrieve the data from the Internet.

If the `useInternet` flag is set to true, then an Internet connection must be established to run the example. Note that the WMS servers may be unavailable, and several minutes may elapse before the maps are returned. One of the challenges of working with WMS servers is that sometimes you will encounter server errors. A function, such as `wmsread`, may time out if a server is unavailable. Often, this is a temporary problem and you will be able to connect to the server if you try again later.

This example writes data to files if `useInternet` is `true` or reads data from files if `useInternet` is `false`. You can store the data locally the first time you run the example and then set the `useInternet` flag to false.

```
useInternet = false;
```

Step 1: Find Katrina Layers From Local Database

One of the more challenging aspects of using WMS is finding a WMS server and then finding the layer that is of interest to you. The process of finding a server that contains the data you need and constructing a specific and often complicated URL with all the relevant details can be very daunting.

The Mapping Toolbox™ simplifies the process of locating WMS servers and layers by providing a local, installed, and pre-qualified WMS database, that is searchable, using the function `wmsfind`. You can search the database for layers and servers that are of interest to you. Here is how you find layers containing the term `katrina` in either the `LayerName` or `LayerTitle` field of the database:

```
katrina = wmsfind('katrina');
whos katrina
```

```
  Name          Size            Bytes  Class       Attributes

  katrina      36x1             17514  WMSLayer
```

The search for the term `'katrina'` returned a `WMSLayer` array containing multiple layers. Extract the first layer.

```
katrina1 = katrina(1);
```

If you type, `katrina`, in the command window, the entire contents of the array are displayed, with each element's index number included in the output. This display makes it easy for you to examine the entire array quickly, searching for a layer of interest. You can display only the `LayerTitle` property for each element by executing the command:

```
disp(katrina,'Properties','layertitle','Index','off','Label','off');
```

As you've discovered, a search for the generic word `'katrina'` returned results of many layers and you need to select only one layer. In general, a search may even return thousands of layers, which may be too large to review individually. Rather than searching the database again, you may refine your search by using the `refine` method of the `WMSLayer` class. Using the `refine` method is more efficient and returns results faster than `wmsfind` since the search has already been narrowed to a smaller set. Supplying the query string, `'goes-12*katrina*visible*close*up*animation'`, to the `refine` method returns a `WMSLayer` array whose elements contain a match of the query string in either the `LayerTitle` or `LayerName` properties. The * character indicates a wild-card search. If multiple entries are returned, select only the first one from the svs.gsfc.nasa.gov server.

```
katrina = refine(katrina,'goes-12*katrina*visible*close*up*animation');
katrina = refine(katrina,'svs.gsfc.nasa.gov','Searchfield','serverurl');
katrina = katrina(1);
whos katrina
```

```
  Name         Size            Bytes  Class       Attributes

  katrina      1x1               466  WMSLayer
```

Step 2: Synchronize WMSLayer Object with Server

The database only stores a subset of the layer information. For example, information from the layer's abstract, details about the layer's attributes and style information, and the coordinate reference system of the layer are not returned by `wmsfind`. To return all the information, you need to use the `wmsupdate` function. `wmsupdate` synchronizes the layer from the database with the server, filling in the missing properties of the layer.

Synchronize the first `katrina` layer with the server in order to obtain the abstract information. Since this action requires access to the Internet, call `wmsupdate` only if the `useInternet` flag is true.

```
if useInternet
    katrina = wmsupdate(katrina);
    save('katrina.mat','katrina')
else
    load('katrina.mat')
end
```

Display the abstract information of the layer. Use `isspace` to help determine where to line wrap the text.

```
abstract = katrina.Abstract;
endOfLine = find(isstrprop(abstract,'cntrl'),1);
abstract = abstract(1:endOfLine);
numSpaces = 60;
while(~isempty(abstract))
    k = find(isspace(abstract));
    n = find(k > numSpaces,1);
    if ~isempty(n)
        fprintf('%s\n',abstract(1:k(n)))
        abstract(1:k(n)) = [];
    else
        fprintf('%s\n',abstract)
        abstract = '';
    end
end
```

```
The GOES-12 satellite sits at 75 degrees west longitude at an
altitude of 36,000 kilometers over the equator, in geosynchronous
orbit.  At this position its Imager instrument takes pictures
of cloud patterns in several wavelengths for all of North and
South America, a primary measurement used in weather forecasting.
 The Imager takes a pattern of pictures of parts of the Earth
in several wavelengths all day, measurements that are vital in
weather forecasting.  This animation shows a daily sequence of
GOES-12 images in the visible wavelengths, from 0.52 to 0.72
microns, during the period that Hurricane Katrina passed through
the Gulf of Mexico.  At one kilometer resolution, the visible
band measurement is the highest resolution data from the Imager,
which accounts for the very high level of detail in these images.
 For this animation, the cloud data was extracted from GOES image
and laid over a background color image of the southeast United

States.
```

Note that this abstract information, including any typographical issues and incomplete fragments, was obtained directly from the server.

Step 3: Explore Katrina Layer Details

You can find out more information about the `katrina` layer by exploring the `Details` property of the `katrina` layer. The `Details.Attributes` field informs you that the layer has fixed width and fixed height attributes, thus the size of the requested map cannot be modified.

```
katrina.Details.Attributes
```

```
ans = struct with fields:
      Queryable: 0
       Cascaded: 0
         Opaque: 1
      NoSubsets: 1
     FixedWidth: 1024
    FixedHeight: 1024
```

The `Details.Dimension` field informs you that the layer has a `time` dimension

```
katrina.Details.Dimension
```

```
ans = struct with fields:
               Name: 'time'
              Units: 'ISO8601'
         UnitSymbol: ''
            Default: '2005-08-30T17:45Z'
    MultipleValues: 0
      NearestValue: 0
           Current: 0
            Extent: '2005-08-23T17:45Z/2005-08-30T17:45Z/P1D'
```

with an extent from `2005-08-23T17:45Z` to `2005-08-30T17:45Z` with a period of `P1D` (one day), as shown in the `Details.Dimension.Extent` field.

```
katrina.Details.Dimension.Extent
```

```
ans =
'2005-08-23T17:45Z/2005-08-30T17:45Z/P1D'
```

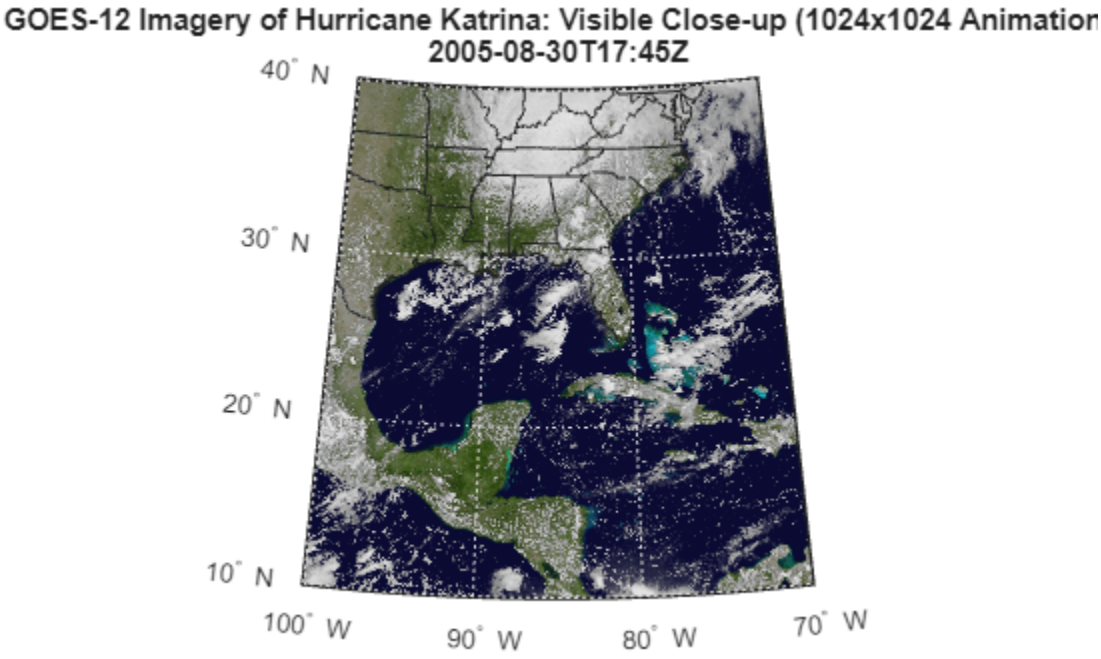
Step 4: Retrieve Katrina Map from Server

Now that you have found a layer of interest, you can retrieve the raster map using the function `wmsread` and display the map using the function `geoshow`. Since `Time` is not specified when reading the layer, the default time, `2005-08-30T17:45Z`, is retrieved as specified by the `Details.Dimension.Default` field. If the `useInternet` flag is set to true, then cache the image and reference object in a GeoTIFF file.

```
if useInternet
    [katrinaMap,R] = wmsread(katrina);
    geotiffwrite('katrina.tif',katrinaMap,R)
else
    [katrinaMap,R] = readgeoraster('katrina.tif');
end
```

Display the `katrinaMap` and overlay the data from the `usastatehi.shp` file.

```
states = readgeotable('usastatehi.shp');
figure
usamap(katrina.Latlim, katrina.Lonlim)
geoshow(katrinaMap,R)
geoshow(states,'FaceColor','none')
title({katrina.LayerTitle, katrina.Details.Dimension.Default}, ...
    'Interpreter','none')
```



Step 5: Find NEXRAD Radar Layer

NEXRAD radar images for the United States are stored on the Iowa State University's IEM Web map server. The server conveniently stores NEXRAD images in five minute increments from `1995-01-01` to the present time. You can find the layer by first searching for the term `IEM WMS Service` in the `ServerTitle` field of the WMS database, then refining the search by requesting the layer of interest, `nexrad-n0r-wmst`.

```
iemLayers = wmsfind('IEM WMS Service','SearchField','servertitle');
nexrad = refine(iemLayers,'nexrad-n0r-wmst');
```

Synchronize the layer with the server.

```
if useInternet
    nexrad = wmsupdate(nexrad);
    save('nexrad.mat','nexrad')
else
    load('nexrad.mat');
end
```

Step 6: Obtain Extent Parameters

To composite the `nexrad` layer with the `katrina` layer, you need to obtain the `nexrad` layer at coincidental time periods, and concurrent geographic and image extents. The `Details.Dimension` field informs you that the layer has a time dimension,

```
nexrad.Details.Dimension
```

```
ans = struct with fields:
              Name: 'time'
             Units: 'ISO8601'
        UnitSymbol: ''
           Default: '2006-06-23T03:10:00Z'
    MultipleValues: 0
      NearestValue: 0
           Current: 0
            Extent: '1995-01-01/2022-12-31/PT5M'
```

and the `Details.Dimension.Default` field informs you that the layer's time extent includes seconds.

```
nexrad.Details.Dimension.Default
```

```
ans =
'2006-06-23T03:10:00Z'
```

Obtain a time value coincidental with the `katrina` layer, and add seconds to the time specification.

```
nexradTime = [katrina.Details.Dimension.Default(1:end-1) ':00Z'];
```

Assign `latlim` and `lonlim` variables to specify the limits for the `nexrad` layer. Set the values to the limits of the `katrina` layer so that the geographic areas match. Note that the `nexrad` layer's southern latitude limit does not extend as far south as the `katrina` layer's southern latitude limit. The values that lie outside the geographic bounding quadrangle of the `nexrad` layer are set to the background color.

```
fprintf('%s%d\n','Southern latitude limit of NEXRAD  layer: ',nexrad.Latlim(1))
```

```
Southern latitude limit of NEXRAD  layer: 24
```

```
fprintf('%s%d\n','Southern latitude limit of Katrina layer: ',katrina.Latlim(1))
```

```
Southern latitude limit of Katrina layer: 10
```

```
latlim = katrina.Latlim;
lonlim = katrina.Lonlim;
```

Assign `imageHeight` and `imageWidth` variables.

```
imageHeight = katrina.Details.Attributes.FixedHeight;
imageWidth  = katrina.Details.Attributes.FixedWidth;
```

Step 7: Retrieve NEXRAD Radar Map from Server

You can retrieve the `nexradMap` from the server, specified at the same time as the `katrinaMap` and for the same geographic and image extents, by supplying parameter/value pairs to the `wmsread` function. To accurately retrieve the radar signal from the map, set the `ImageFormat` parameter to the `image/png` format. In order to easily retrieve the signal from the background, set the background color to black (`[0 0 0]`).

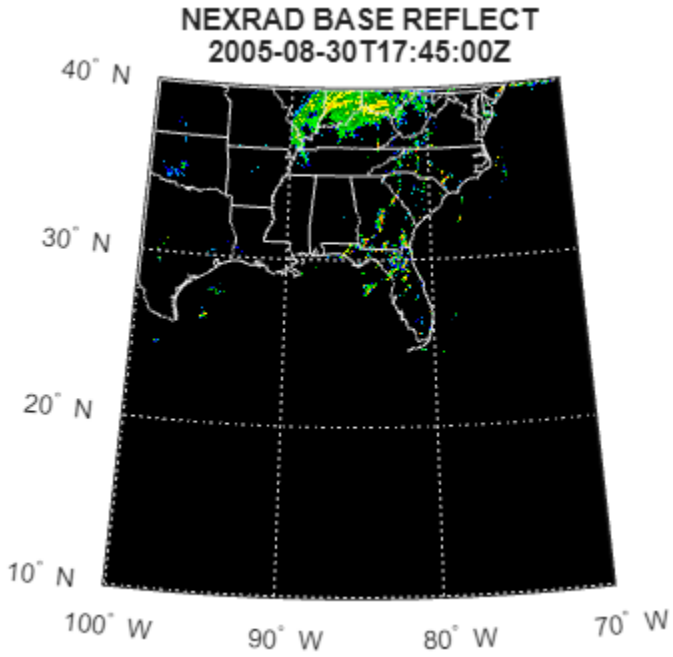
Retrieve the `nexradMap`.

```
black = [0 0 0];
if useInternet
```

```
    [nexradMap,R] = wmsread(nexrad, ...
        'Latlim',latlim,'Lonlim',lonlim,'Time',nexradTime, ...
        'BackgroundColor',black,'ImageFormat','image/png', ...
        'ImageHeight',imageHeight,'ImageWidth',imageWidth);
    geotiffwrite('nexrad.tif',nexradMap,R)
else
    [nexradMap,R] = readgeoraster('nexrad.tif');
end
```

Display the `nexradMap`.

```
figure
usamap(latlim,lonlim)
geoshow(nexradMap,R)
geoshow(states,'FaceColor','none','EdgeColor',[0.9 0.9 0.9])
title({nexrad.LayerTitle, nexradTime},'Interpreter','none');
```



Step 8: Composite NEXRAD Radar Map with Katrina Map

To composite the `nexradMap` with a copy of the `katrinaMap`, you need to identify the non-background pixels in the `nexradMap`. The `nexradMap` data is returned as an image with class double, because of how this web map server handles `PNG` format, so you need convert it to `uint8` before merging.

Identify the pixels of the `nexradMap` image that do not contain the background color.

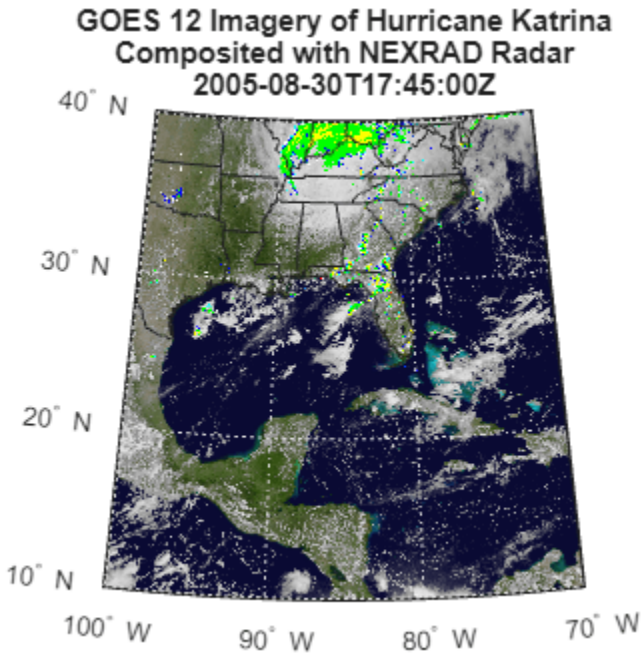
```
threshold = 0;
index = any(nexradMap > threshold, 3);
index = repmat(index,[1 1 3]);
```

Composite the `nexradMap` with the `katrinaMap`.

```
combination = katrinaMap;
combination(index) = uint8(nexradMap(index)*255);
```

Display the composited map.

```
figure
usamap(latlim,lonlim)
geoshow(combination,R)
geoshow(states,'FaceColor','none')
title({'GOES 12 Imagery of Hurricane Katrina', ...
    'Composited with NEXRAD Radar',nexradTime})
```



Step 9: Initialize Variables to Animate the Katrina and NEXRAD Maps

The next step is to initialize variables in order to animate the composited `katrina` and `nexrad` maps.

Create variables that contain the time extent of the `katrina` layer.

```
extent = katrina.Details.Dimension.Extent;
slash = '/';
slashIndex = strfind(extent,slash);
startTime = extent(1:slashIndex(1)-1);
endTime = extent(slashIndex(1)+1:slashIndex(2)-1);
```

Calculate numeric values for the start and end days. Note that the time extent is in `yyyy-mm-dd` format.

```
hyphen = '-';
hyphenIndex = strfind(startTime,hyphen);
dayIndex = [hyphenIndex(2) + 1, hyphenIndex(2) + 2];
startDay = str2double(startTime(dayIndex));
endDay = str2double(endTime(dayIndex));
```

Assign the initial `katrinaTime`.

```
katrinaTime = startTime;
```

Since multiple requests to a server are required for animation, it is more efficient to use the `WebMapServer` and `WMSMapRequest` classes.

Construct a `WebMapServer` object for each layer's server.

```
nasaServer = WebMapServer(katrina.ServerURL);
iemServer  = WebMapServer(nexrad.ServerURL);
```

Create `WMSMapRequest` objects.

```
katrinaRequest = WMSMapRequest(katrina, nasaServer);
nexradRequest  = WMSMapRequest(nexrad, iemServer);
```

Assign properties.

```
nexradRequest.Latlim = latlim;
nexradRequest.Lonlim = lonlim;
nexradRequest.BackgroundColor = black;
nexradRequest.ImageFormat = 'image/png';
nexradRequest.ImageHeight = imageHeight;
nexradRequest.ImageWidth  = imageWidth;
```

Step 10: Create Animation Files

An animation can be viewed in the browser when the browser opens an animated GIF file or an AVI video file. To create the animation frames of the WMS basemap and vector overlays, create a loop through each day, from `startDay` to `endDay`, and obtain the `katrinaMap` and the `nexradMap` for that day. Composite the maps into a single image, display the image, retrieve the frame, and store the results into a frame of an AVI file and a frame of an animated GIF file.

To share with others or to post to web video services, create an AVI video file containing all the frames using the `VideoWriter` class.

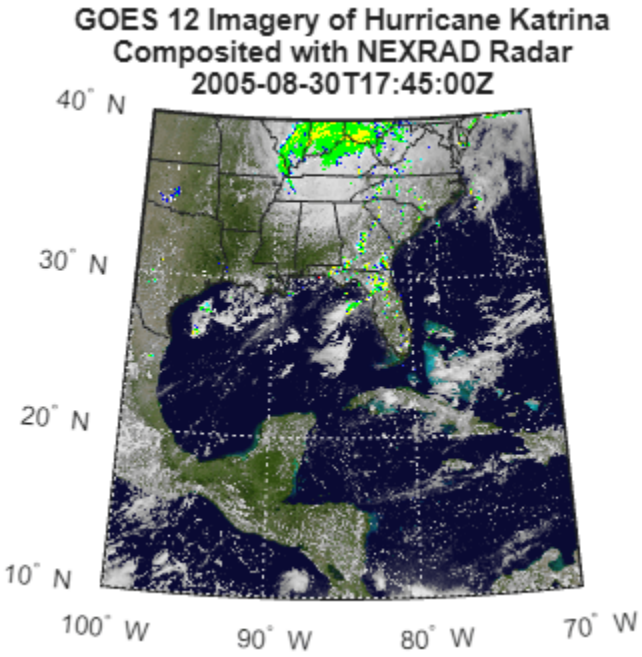
```
videoFilename = fullfile(pwd,'wmsanimated.avi');
if exist(videoFilename,'file')
    delete(videoFilename)
end
writer = VideoWriter(videoFilename);
writer.FrameRate = 1;
writer.Quality = 100;
open(writer)
```

The animation is viewed in a single map display. Outside the animation loop, create a map display. Initialize `hmap`, used in the loop as the return handle from the function `geoshow`, so it can be deleted on the first pass through the loop. Loop through each day, retrieve and display the WMS map, and save the frame.

```
fig = figure;
usamap(latlim,lonlim)
hstates = geoshow(states,'FaceColor','none');
hmap = [];

for k = startDay:endDay

    % Update the time values and assign the Time property for each server.
    currentDay = num2str(k);
    katrinaTime(dayIndex) = currentDay;
    nexradTime = [katrinaTime(1:end-1) ':00Z'];
```

```
katrinaRequest.Time = katrinaTime;
nexradRequest.Time = nexradTime;

% Retrieve the WMS map of Katrina from the server (or file)
% for this time period.
nmKatrina = ['katrina_' num2str(currentDay) '.tif'];
if useInternet
    katrinaMap = getMap(nasaServer, katrinaRequest.RequestURL);
    geotiffwrite(nmKatrina, katrinaMap, katrinaRequest.RasterRef)
else
    katrinaMap = readgeoraster(nmKatrina);
end

% Retrieve the WMS map of the NEXRAD imagery from the server (or file)
% for this time period.
nmNEXRAD = ['nexrad_' num2str(currentDay) '.tif'];
if useInternet
    nexradMap = getMap(iemServer, nexradRequest.RequestURL);
    geotiffwrite(nmNEXRAD, nexradMap, nexradRequest.RasterRef)
else
    nexradMap = readgeoraster(nmNEXRAD);
end

% Identify the pixels of the nexradMap image that do not contain the
% background color.
index = any(nexradMap > threshold, 3);
index = repmat(index,[1 1 3]);

% Composite nexradMap with katrinaMap.
combination = katrinaMap;
combination(index) = uint8(nexradMap(index)*255);

% Delete the old map and display the new composited map.
delete(hmap)
hmap = geoshow(combination, katrinaRequest.RasterRef);
uistack(hstates,'top')
title({'GOES 12 Imagery of Hurricane Katrina', ...
    'Composited with NEXRAD Radar',nexradTime})
drawnow

% Save the current frame as an RGB image.
currentFrame = getframe(fig);
RGB = currentFrame.cdata;

% Create an indexed image for each RGB frame in order to display an
% animated GIF.
if k == startDay
    % The first time through the loop, convert the RGB image to
    % an indexed image and save the colormap into the
    % variable, cmap. Use cmap to convert later frames.
    [frame,cmap] = rgb2ind(RGB,256,'nodither');

    % Use the size of the first frame and the total
    % number of frames to initialize animated with
    % a size large enough to contain all the frames.
    frameSize = size(frame);
    numFrames = endDay - startDay + 1;
    animated = zeros([frameSize 1 numFrames],'like',frame);
```

```
    else
        % Use the colormap from the first frame conversion and
        % convert this frame to an indexed image.
        frame = rgb2ind(RGB,cmap,'nodither');
    end

    % Store the frame into the animated array for the GIF file.
    frameCount = k - startDay + 1;
    animated(:,:,1,frameCount) = frame;

    % Write the RGB frame to the AVI file.
    writeVideo(writer,RGB);
end
```



Close the Figure window and the AVI file.

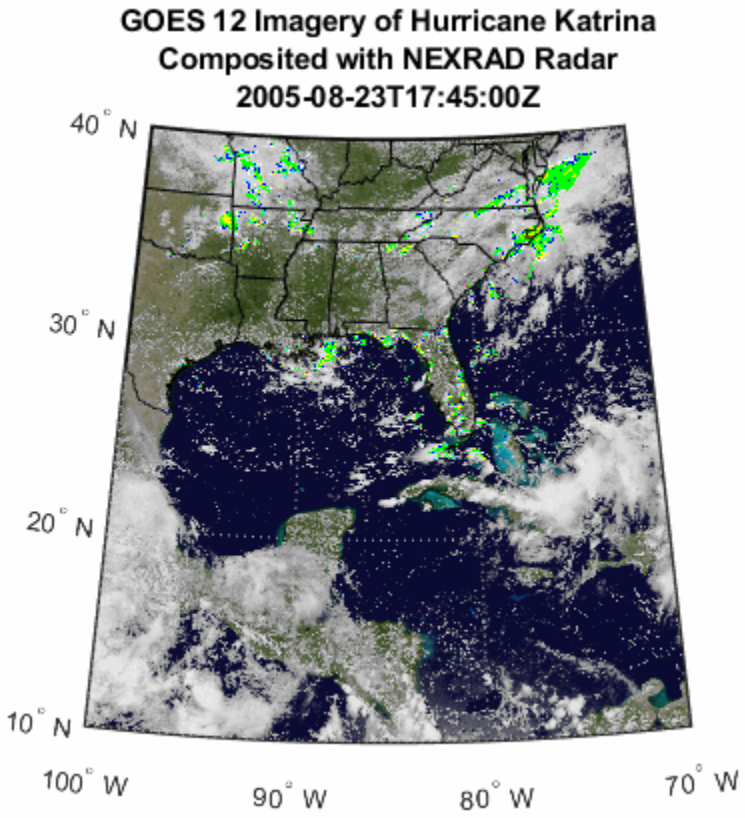
```
close(fig)
close(writer)
```

Write the animated GIF file.

```
filename = 'wmsanimated.gif';
delayTime = 2.0;
loopCount = inf;
imwrite(animated,cmap,filename, ...
    'DelayTime',delayTime,'LoopCount',loopCount);
```

Step 11: View Animated GIF File

An animation can be viewed in the browser when the browser opens an animated GIF file.



Credits

Katrina Layer

The Katrina layer used in the example is from the NASA Goddard Space Flight Center's SVS Image Server and is maintained by the Scientific Visualization Studio.

For more information about this server, use the command `wmsinfo('http://svs.gsfc.nasa.gov/cgi-bin/wms?')`.

NEXRAD Layer

The NEXRAD layer used in the example is from the Iowa State University's IEM WMS server and is a generated CONUS composite of National Weather Service (NWS) WSR-88D level III base reflectivity.

For more information about this server, use the command `wmsinfo('http://mesonet.agron.iastate.edu/cgi-bin/wms/nexrad/n0r-t.cgi?')`.

See Also

Functions
`wmsfind` | `refine` | `wmsupdate` | `wmsread` | `usamap` | `geoshow`

Objects
`WMSMapRequest` | `WebMapServer`

Web Map Migration Strategies

Web maps, including functions such as `webmap`, `wmmarker`, and `wmline`, will be removed in a future release. Use geographic axes instead. This topic provides strategies for updating your code to use geographic axes.

Benefits of Geographic Axes

Unlike web maps, geographic axes enable you to:

- Display data using additional plot types, including density plots and bubble charts. For examples, see “Create Maps Using Latitude and Longitude Data”.
- Customize your maps using MATLAB graphics functions. For examples, see “Add Labels and Annotations to Geographic Axes” and “Customize Appearance of Geographic Axes”.
- Build apps using App Designer. For an example, see “Create Geographic Axes in App”.

Replacement Functions

These web map functions will be removed in a future release. To update your code, replace uses of the web map functions with geographic axes functions.

To Be Removed	Recommended
`webmap`	Use a geographic axes object. • In most cases, create a geographic axes object by using a plotting function such as `geoplot` or `geobasemap`. • When you want advanced control of the geographic axes object, create the object by using the `geoaxes` function. For detailed replacement patterns, see the “Version History” section of the `webmap` page.
`wmmarker`	Use the `geoiconchart` function. For detailed replacement patterns, see the “Version History” section of the `wmmarker` page.
`wmline`	Use the `geoplot` function. For detailed replacement patterns, see the “Version History” section of the `wmline` page.
`wmpolygon`	Use a `geopolyshape` object and the `geoplot` function. For detailed replacement patterns, see the “Version History” section of the `wmpolygon` page.

To Be Removed	Recommended
`wmremove`	Use the `delete` function. For detailed replacement patterns, see the "Version History" section of the `wmremove` page.
`wmcenter`	Use the `MapCenter` property of the geographic axes object. For detailed replacement patterns, see the "Version History" section of the `wmcenter` page.
`wmzoom`	Use the `ZoomLevel` property of the geographic axes object. For detailed replacement patterns, see the "Version History" section of the `wmzoom` page.
`wmlimits`	Use the `geolimits` function instead. For detailed replacement patterns, see the "Version History" section of the `wmlimits` page.
`wmclose`	Use the `close` function. For detailed replacement patterns, see the "Version History" section of the `wmclose` page.
`wmprint`	Use an export option from "Compare Ways to Export Graphics from Figures".

Update Code to Use Geographic Axes

In many cases, you can update your code to use geographic axes by following these steps:

1. Add base layers that are not available as predefined basemaps by using the `addCustomBasemap` function.
2. Replace uses of the `webmap` function with a function that creates a geographic axes object. For example, you can create a geographic axes object with a specified basemap by using the `geobasemap` function.
3. If you want to display multiple graphics objects in the same geographic axes object, set the hold state to `on` by using the `hold` function.
4. Replace uses of the `wmmarker` function with the `geoiconchart` function.
5. Replace uses of the `wmline` function with the `geoplot` function.
6. Replace uses of the `wmpolygon` function with a `geopolyshape` object and the `geoplot` function.
7. Replace uses of the `wmlimits` function with the `geolimits` function.

The steps in this table create similar maps using web map functions and geographic axes functions.

Step	Web Map (To Be Removed)	Geographic Axes (Recommended)

Add base layers that are not available as predefined basemaps.	Not applicable.	Add a custom basemap from OpenStreetMap by using the `addCustomBasemap` function.
		
		`basemapName = "openstreetmap";`
		`url = "a.tile.openstreetmap.org/${z}/${`
		`attribution = char(uint8(169)) + "OpenS`
		`addCustomBasemap(basemapName,url,Attrib`

Create a map that uses a specified basemap.	Create a web map that uses the OpenStreetMap basemap by using the `webmap` function.
	
	`webmap("OpenStreetMap")`
		Create a geographic axes object that uses the OpenStreetMap basemap by using the `geobasemap` function.
		
		`figure`
		`geobasemap openstreetmap`

Prepare to display multiple graphics objects on the same map.	Not applicable.	Prepare to add additional data to the map by setting the hold state of the axes to `on`. `hold on`
Display a marker on the map.	Display a marker by using the `wmmarker` function. `wmmarker(42.3557,-71.0662)`	Display an icon by using the `geoiconchart` function. Increase the size of the icon by specifying the `SizeData` argument. `geoiconchart(42.3557,-71.0662,SizeData=`

Display a line on the map.	Specify the coordinates of a line. Display the line by using the `wmline` function.
	
	`lineLat = [42.3536 42.3548 42.3561 42.3573];`
	`lineLon = [-71.0656 -71.0665 -71.0632 -71.0637];`
	
	`wmline(lineLat,lineLon)`
		Specify the coordinates of a line. Display the line by using the `geoplot` function. Specify the line width and color by using the `LineWidth` and `Color` arguments.
		
		`lineLat = [42.3536 42.3548 42.3561 42.3`
		`lineLon = [-71.0656 -71.0665 -71.0632 -`
		
		`geoplot(lineLat,lineLon,LineWidth=2,Col`

Display a polygon on the map.	Specify the latitude and longitude coordinates of the polygon. Display the polygon by using the `wmpolygon` function.
	
	`polyLat = [42.3554 42.3560 42.3527 42.3520 42.3554];`
	`polyLon = [-71.0722 -71.0695 -71.0678 -71.0707 -71.0722];`
	
	`wmpolygon(polyLat,polyLon,FaceAlpha=0.35)`
		Create a polygon shape from latitude and longitude coordinates by using the `geopolyshape` function. Display the polygon by using the `geoplot` function.
		
		`polyLat = [42.3554 42.3560 42.3527 42.3`
		`polyLon = [-71.0722 -71.0695 -71.0678 -`
		`shape = geopolyshape(polyLat,polyLon);`
		
		`geoplot(shape,FaceColor=[0 0 0])`

Step	Web Map (To Be Removed)	Geographic Axes (Recommended)

Change the geographic limits of the map.	Specify new latitude and longitude limits. Change the limits of the map by using the `wmlimits` function.
	
	`latlim = [42.3518 42.3577];`
	`lonlim = [-71.0735 -71.0609];`
	`wmlimits(latlim,lonlim)`
		Specify new latitude and longitude limits. Change the limits of the map by using the `geolimits` function.
		
		`latlim = [42.3518 42.3577];`
		`lonlim = [-71.0735 -71.0609];`
		`geolimits(latlim,lonlim)`

View the map.		

Interact with Geographic Axes

Like web maps, you can interactively explore geographic axes by panning and zooming. Pan by dragging or by using the arrow keys. Zoom by scrolling or by using the **Plus** and **Minus** keys. For more information about controlling the interactivity of axes objects, see “Control Chart Interactivity”.

By default, geographic axes do not include interactive controls for selecting a basemap, navigating the map, or identifying the coordinates of the pointer. For an example that shows how to add interactive controls to geographic axes, see “Add Interactive Controls to Geographic Axes” on page 7-80.

See Also

Functions

`addCustomBasemap` | `geobasemap` | `geoiconchart` | `geoplot` | `geolimits`

Properties
GeographicAxes Properties

Related Examples

- “Use Base Layers in Geographic Axes” on page 7-75
- “Add Interactive Controls to Geographic Axes” on page 7-80

Use Base Layers in Geographic Axes

Web maps, including functions such as `webmap`, `wmmarker`, and `wmline`, will be removed in a future release. Use geographic axes instead.

This topic provides strategies for using base layers (also called basemaps) in geographic axes. For some base layers, you can update your code by using a predefined basemap. For other base layers, you can update your code by using a custom basemap.

Benefits of Geographic Axes

Both web maps and geographic axes enable you to display custom basemaps created from raster map tiles. Unlike web maps, geographic axes also enable you to:

- Display custom basemaps from vector map tiles hosted by Esri. For an example, see “Add Vector Basemap from Esri”.
- Display custom basemaps from vector or raster MBTiles files. For an example, see “Add Basemap from MBTiles File”.
- Display custom basemaps created from georeferenced images or data grids. For examples, see “Add Basemap from Image” and “Add Basemap from Quantitative Data”.

Add custom basemaps by using the `addCustomBasemap` function. After you add a custom basemap, you can display the basemap in a geographic axes object by using the `geobasemap` function. Alternatively, display the basemap by setting the `Basemap` property of the geographic axes object.

Replace Base Layers

This table lists the web map base layer options that are available in geographic axes as predefined basemaps. For a full list of predefined basemaps, see the `basemap` argument of the `geobasemap` function.

Base Layer (To Be Removed)	Predefined Basemap (Recommended)
`"World Street Map"`	Use the `"streets"` basemap. `geobasemap streets`
`"World Imagery"`	Use the `"satellite"` basemap. `geobasemap satellite`
`"Light Gray Canvas Map"`	Use the `"streets-light"` basemap. `geobasemap streets-light`

This table lists the base layer options that are not available in geographic axes as predefined basemaps. For these base layers, add a custom basemap by using the `addCustomBasemap` function with a map tiles URL and an attribution. You can optionally specify a display name, as well. For each base layer option, the table includes the map tiles URL, the attribution, and a display name.

Note Custom basemaps added from map tile URLs depend on the stability of the URL. If a map tile URL becomes unavailable, then the custom basemap can be slow to display, partially display, or fail to display.

Base Layer (To Be Removed)	Custom Basemap (Recommended)

"Open Street Map"	Add the custom basemap.
	
	`url = "https://a.tile.openstreetmap.org/${z}/${x}/${y`
	`attrib = char(uint8(169)) + " OpenStreetMap contribut`
	`addCustomBasemap("openstreetmap",url,Attribution=attr`
	`  DisplayName="OpenStreetMap")`
	
	Use the custom basemap.
	
	`geobasemap openstreetmap`

"World Topographic Map"	Add the custom basemap.
	
	`url = "https://server.arcgisonline.com/ArcGIS/rest/se`
	`  "World_Topo_Map/MapServer/tile/${z}/${y}/${x}.png`
	`attrib = "Sources: Esri, HERE, Garmin, Intermap, incr`
	`  "GEBCO, USGS, FAO, NPS, NRCAN, GeoBase, IGN, Kada`
	`  "Ordnance Survey, Esri Japan, METI, Esri China (H`
	`  "(c) OpenStreetMap contributors, and the GIS User`
	`addCustomBasemap("worldtopomap",url,Attribution=attri`
	`  DisplayName="World Topographic Map")`
	
	Use the custom basemap.
	
	`geobasemap worldtopomap`

"World Shaded Relief"	Add the custom basemap.
	
	`url = "https://server.arcgisonline.com/ArcGIS/rest/se`
	`  "World_Shaded_Relief/MapServer/tile/${z}/${y}/${x`
	`attrib = "Copyright:(c) 2014 Esri";`
	`addCustomBasemap("worldshadedrelief",url,Attribution=`
	`  DisplayName="World Shaded Relief")`
	
	Use the custom basemap.
	
	`geobasemap worldshadedrelief`

"World Physical Map"	Add the custom basemap.
	
	`url = "https://server.arcgisonline.com/ArcGIS/rest/se`
	`  "World_Physical_Map/MapServer/tile/${z}/${y}/${x}`
	`attrib = "Source: US National Park Service";`
	`addCustomBasemap("worldphysicalmap",url,Attribution=a`
	`  DisplayName="World Physical Map")`
	
	Use the custom basemap.
	
	`geobasemap worldphysicalmap`

Base Layer (To Be Removed)	Custom Basemap (Recommended)
"World Terrain Base"	Add the custom basemap. url = "https://server.arcgisonline.com/ArcGIS/rest/se "World_Terrain_Base/MapServer/tile/${z}/${y}/${x} attrib = "Sources: Esri, USGS, NOAA"; addCustomBasemap("worldterrainbase",url,Attribution=a DisplayName="World Terrain Base") Use the custom basemap. geobasemap worldterrainbase
"USGS Imagery"	Add the custom basemap. url = "https://basemap.nationalmap.gov/ArcGIS/rest/se "USGSImageryOnly/MapServer/tile/${z}/${y}/${x}/pn attrib = "USDA, USGS The National Map: Orthoimagery." addCustomBasemap("usgsimagery",url,Attribution=attrib DisplayName="USGS Imagery") Use the custom basemap. geobasemap usgsimagery
"USGS Topographic Imagery"	Add the custom basemap. url = "https://basemap.nationalmap.gov/ArcGIS/rest/se "USGSImageryTopo/MapServer/tile/${z}/${y}/${x}/pn attrib = "USGS The National Map: Orthoimagery and US addCustomBasemap("usgsimagerytopo",url,Attribution=at DisplayName="USGS Topographic Imagery") Use the custom basemap. geobasemap usgsimagerytopo
"USGS Shaded Topographic Map"	Add the custom basemap. url = "https://basemap.nationalmap.gov/ArcGIS/rest/se "USGSTopo/MapServer/tile/${z}/${y}/${x}/png"; attrib = "USGS The National Map: National Boundaries "3DEP Elevation Program, Geographic Names Informa "National Hydrography Dataset, National Land Cove "National Structures Dataset, and National Transp "USGS Global Ecosystems; U.S. Census Bureau TIGER "USFS Road data; Natural Earth Data; U.S. Departm "NOAA National Centers for Environmental Informat addCustomBasemap("usgstopo",url,Attribution=attrib, . DisplayName="USGS Shaded Topographic Map") Use the custom basemap. geobasemap usgstopo

Base Layer (To Be Removed)	Custom Basemap (Recommended)

`"National Geographic Map"`	Add the custom basemap.
	
	`url = "https://server.arcgisonline.com/ArcGIS/rest/se`
	`    "NatGeo_World_Map/MapServer/tile/${z}/${y}/${x}.p`
	`attrib = "National Geographic, Esri, Garmin, HERE, UN`
	`    "USGS, NASA, ESA, METI, NRCAN, GEBCO, NOAA, incre`
	`addCustomBasemap("natgeomap",url,Attribution=attrib,`
	`    DisplayName="National Geographic Map")`
	
	Use the custom basemap.
	
	`geobasemap natgeomap`

`"Ocean Basemap"`	Add the custom basemap.
	
	`url = "https://server.arcgisonline.com/ArcGIS/rest/se`
	`    "Ocean/World_Ocean_Base/MapServer/tile/${z}/${y}/`
	`attrib = "Esri, Garmin, GEBCO, NOAA NGDC, and other c`
	`addCustomBasemap("oceanbasemap",url,Attribution=attri`
	`    DisplayName="Ocean Basemap")`
	
	Use the custom basemap.
	
	`geobasemap oceanbasemap`

`"World Navigation Charts"`	Add the custom basemap.
	
	`url = "https://server.arcgisonline.com/ArcGIS/rest/se`
	`    "Specialty/World_Navigation_Charts/MapServer/tile`
	`attrib = "Copyright:(c) 2013 East View Cartographic";`
	`addCustomBasemap("worldnavcharts",url,Attribution=att`
	`    DisplayName="World Navigation Charts")`
	
	Use the custom basemap.
	
	`geobasemap worldnavcharts`

Interactively Select Basemaps

By default, geographic axes do not include interactive controls for selecting a basemap. For an example that shows how to add a basemap picker to a geographic axes object, see “Add Interactive Controls to Geographic Axes” on page 7-80.

See Also

Functions
`geobasemap` | `addCustomBasemap`

Properties
GeographicAxes Properties

Related Examples

- “Web Map Migration Strategies” on page 7-70

- “Add Interactive Controls to Geographic Axes” on page 7-80

Add Interactive Controls to Geographic Axes

Typically, you control the appearance of geographic axes by using default pan and zoom interactions, by using functions such as `geobasemap` and `geolimits`, and by changing the properties of `GeographicAxes` objects.

You can also control the appearance of geographic axes by adding interactive controls. This topic shows how to create a geographic axes object that contains a basemap picker, a zoom slider, and a context menu that displays and copies the coordinates of the pointer.

For information about creating apps with similar interactive controls in App Designer, see “Create Geographic Axes in App”.

Add Basemap Picker to Geographic Axes

Add a basemap picker to the axes toolbar of a geographic axes object by using the `addToolbarMapButton` function.

Display the locations of European capitals on a map. When there is no current axes, the `geoiconchart` function plots the data by creating a new geographic axes object.

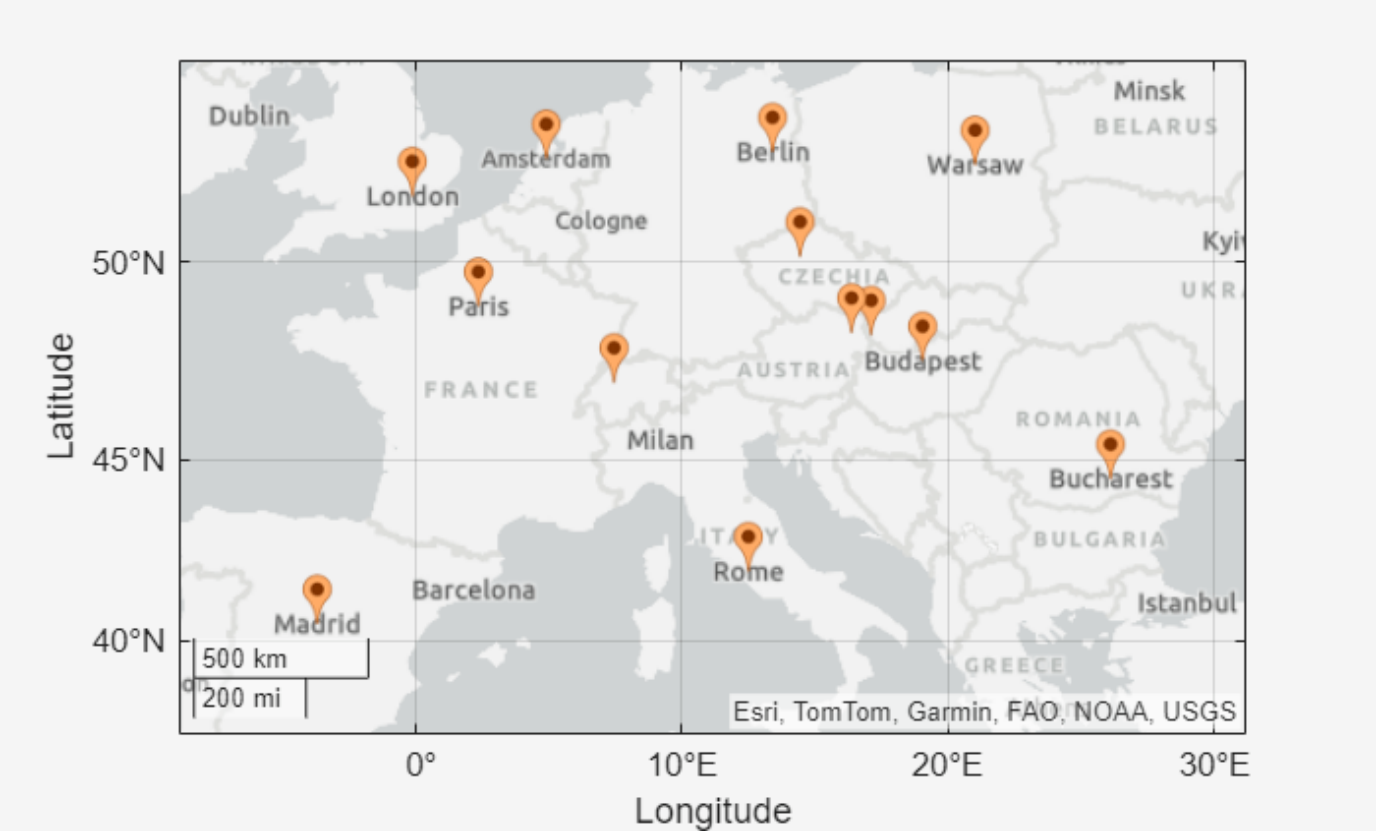
```
[lat,lon] = readvars("european_capitals.txt");

figure
geoiconchart(lat,lon)
geolimits([37.3 54.6],[-8.9 31.2])
```

Preserve the toolbar, limits, and plotted data by using the `hold` function.

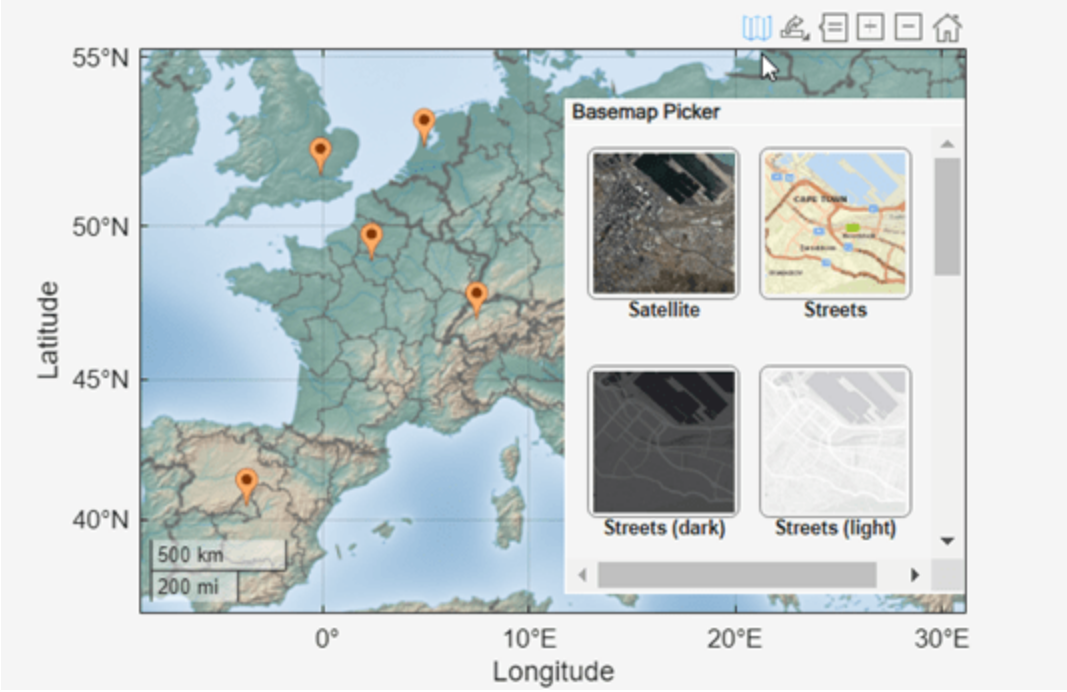
```
hold on
```

Create a custom toolbar by using the `axtoolbar` function. Add the basemap picker to the toolbar.

```
tb = axtoolbar("default");
addToolbarMapButton(tb,"basemap")
```



Open the basemap picker and select a basemap.



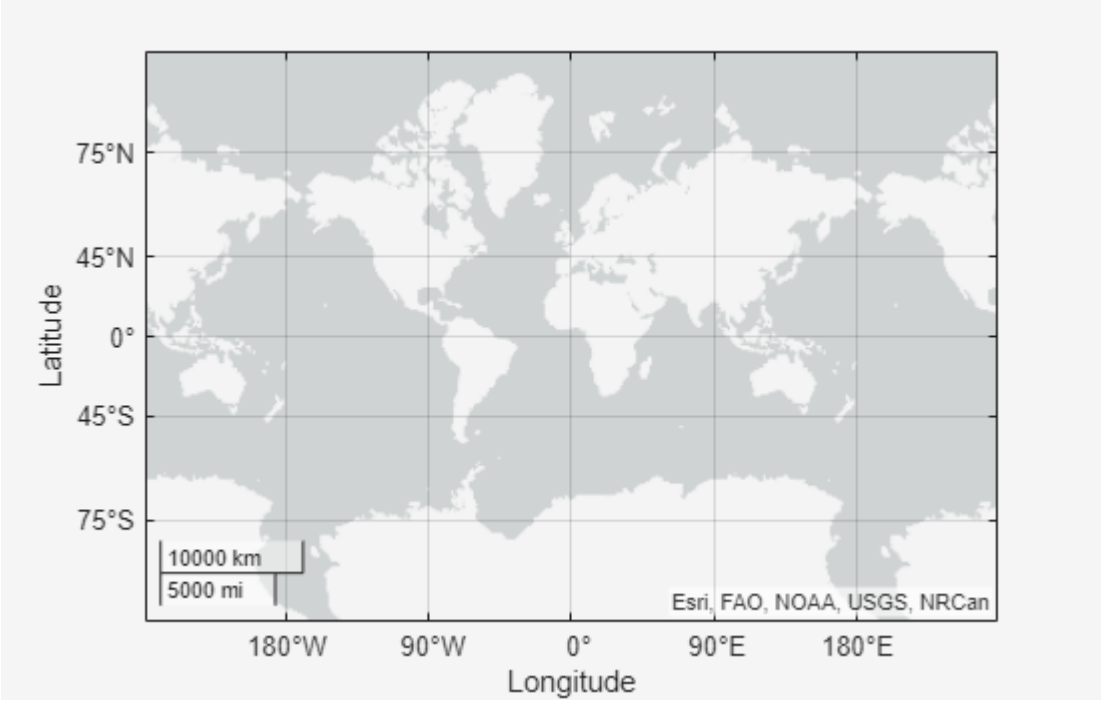
By default, the `addToolbarMapButton` function adds all available basemaps to the basemap picker. For information about customizing the appearance of the basemap picker, see the `addToolbarMapButton` page.

Add Zoom Slider to Geographic Axes

Add a zoom slider to a geographic axes object by using a slider component. Create the slider component by using the `uislider` function.

Sliders created with `uislider` are typically used for app development, so create a figure by using the `uifigure` function. Specify the width of the figure as 550 pixels and the height of the figure as 350 pixels. Then, add a geographic axes object to the figure.

```
uif = uifigure;
uif.Position(3:4) = [550 350];

gx = geoaxes(uif);
```



Add a slider component to the figure. Customize the slider by changing properties of the `Slider` object.

- Position the slider on the right side of the map.
- Use a vertical orientation instead of the default horizontal orientation.
- Set the value of the slider by querying the zoom level of the map.
- Set the range of the slider to [0, 18].
- Program the behavior of the zoom slider using an anonymous callback function. For more information about callbacks, see “Create Callbacks for Apps Created Programmatically”.

```
sld = uislider(uif);

sld.Position(1:2) = [510 100];
sld.Orientation = "vertical";
sld.Value = gx.ZoomLevel;
sld.Limits = [0 18];
sld.ValueChangedFcn = @(src,event)set(gx,"ZoomLevel",event.Value);
```

Display a marker at the address of the Eiffel Tower.

```
GT = geocode("Av. Gustave Eiffel, 75007 Paris, France","address");
geoiconchart(gx,GT.Shape.Latitude,GT.Shape.Longitude)
```

Zoom in by moving the slider thumb. As you move the thumb, the zoom level of the map changes to reflect the slider value.

For more information about creating and customizing slider components, see `uislider` and `Slider`.

Get Coordinates of Pointer

Display the coordinates of the pointer, and copy the coordinates to the clipboard by using a context menu.

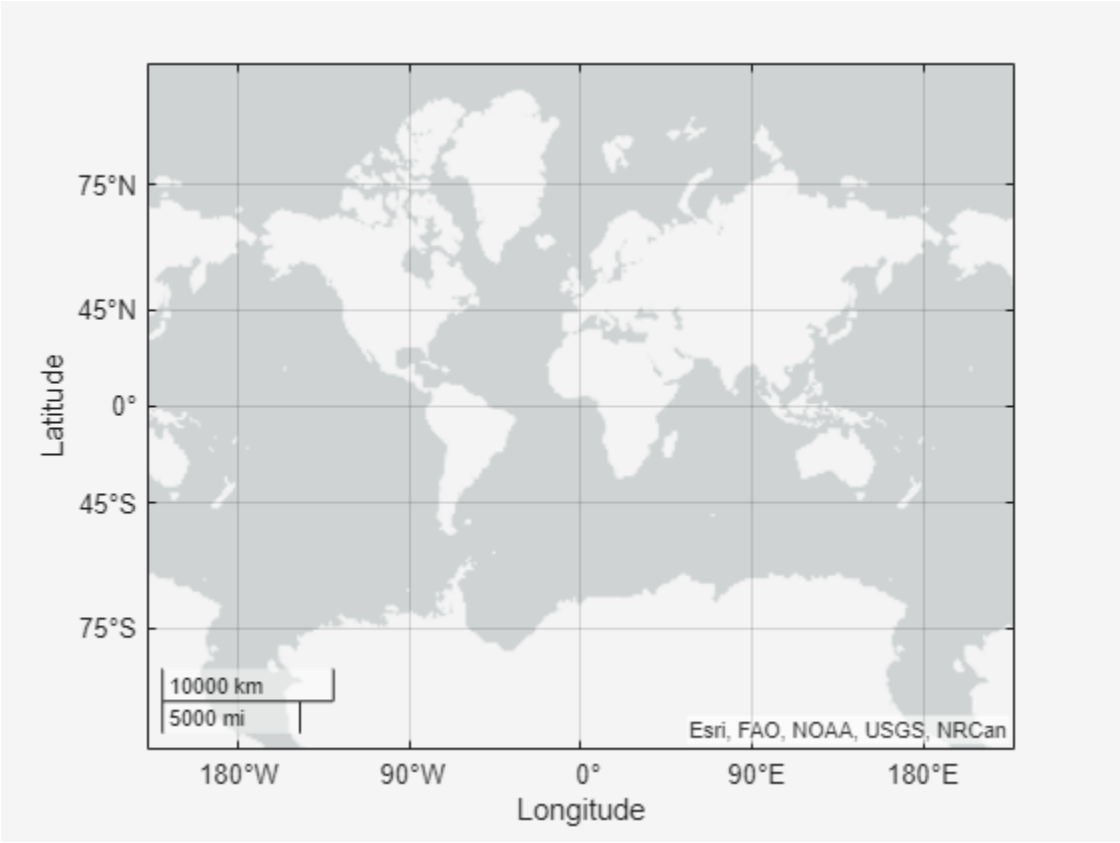
Context menus are typically used for app development, so create a figure by using the `uifigure` function. Then, add a geographic axes object to the figure.

```
uif = uifigure;
gx = geoaxes(uif);
```

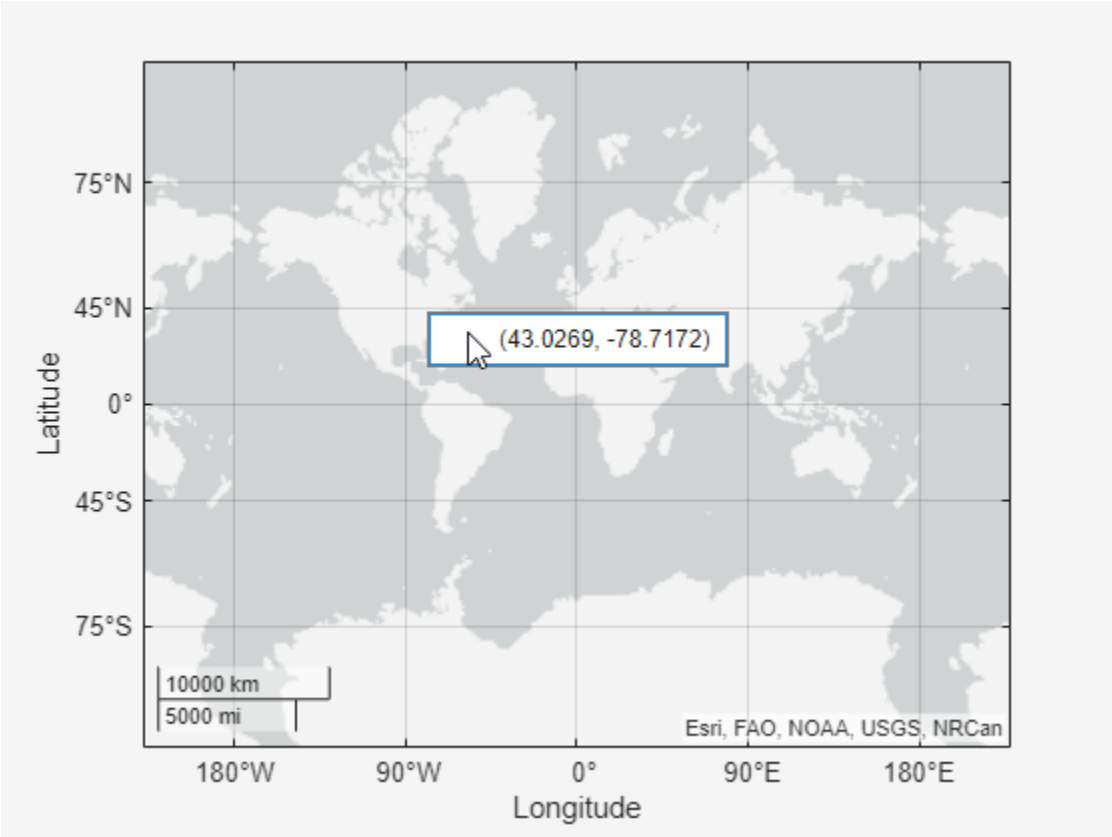
Create a context menu with one menu item. Define the behavior of the context menu and menu item using callback functions. For more information about callback functions, see “Create Callbacks for Apps Created Programmatically”.

- Add the context menu to the geographic axes object. Specify a `ContextMenuOpeningFcn` callback function using the `updateCoordinateDisplay` on page 7-86 local function. When you right-click the map, the function displays the coordinates of the pointer.
- Add the menu item to the context menu. Specify a `MenuSelectedFcn` callback function using the `copyCoordinatesToClipboard` on page 7-86 local function. When you click the menu item, the function copies the coordinates to the clipboard.

```
gx.ContextMenu = uicontextmenu(uif, ...
    ContextMenuOpeningFcn = @(src,event)updateCoordinateDisplay(gx,event));

m = uimenu(gx.ContextMenu, ...
    MenuSelectedFcn = @(src,event)copyCoordinatesToClipboard(gx));
```



Display the coordinates of the pointer by right-clicking the map. Copy the coordinates to the clipboard by clicking the menu item.



For more information about creating context menus and menu items, see `uicontextmenu` and `uimenu`.

Local Functions

Define the `updateCoordinateDisplay` and `copyCoordinatesToClipboard` functions, used as callback functions in this example. Both functions use the `CurrentPoint` property of the input geographic axes `gx` to get the geographic coordinates of the pointer. The `updateCoordinateDisplay` function uses the `event.Source.Children` property to access the menu item stored in the context menu.

```
function updateCoordinateDisplay(gx,event)
    pt = gx.CurrentPoint;
    coordinateString = "(" + pt(1,1) + ", " + pt(1,2) + ")";
    set(event.Source.Children,Text=coordinateString)
end

function copyCoordinatesToClipboard(gx)
    pt = gx.CurrentPoint;
    coordinateString = "(" + pt(1,1) + ", " + pt(1,2) + ")";
```

```
    clipboard("copy",coordinateString)
end
```

See Also

Functions
`geoaxes` | `addToolbarMapButton` | `uislider` | `uicontextmenu` | `uimenu`

Properties
GeographicAxes Properties | Slider | ContextMenu | Menu Properties

Related Examples

- “Create Callbacks for Apps Created Programmatically”
- “Create Geographic Axes in App”

Geometric Geodesy

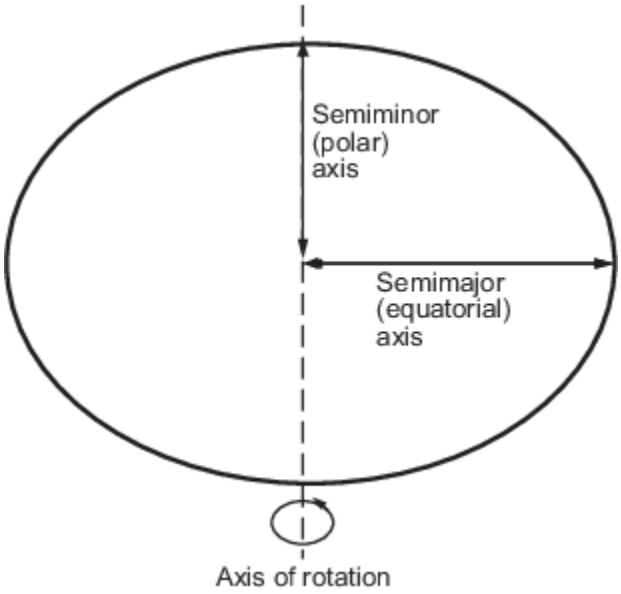
- “Shape of the Earth” on page 8-2
- “Comparison of Reference Spheroids” on page 8-4
- “Work with Reference Spheroids” on page 8-11
- “Comparison of 3-D Coordinate Systems” on page 8-13
- “Vectors in 3-D Coordinate Systems” on page 8-18
- “Plotting a 3-D Dome as a Mesh over a Globe” on page 8-21
- “Find Ellipsoidal Height from Orthometric Height” on page 8-27
- “Angle Representations and Angular Units” on page 8-30
- “Length and Distance Units” on page 8-33
- “Distances on the Sphere” on page 8-35
- “Comparison of Rhumb Lines and Great Circles” on page 8-38
- “Generate Small Circles” on page 8-40
- “Interactively Create Small Circle and Track Annotations on Maps” on page 8-42

Shape of the Earth

Although the Earth is very round, it is an oblate spheroid rather than a perfect sphere. This difference is so small (only one part in 300) that modeling the Earth as spherical is sufficient for making small-scale (world or continental) maps. However, making accurate maps at larger scale demands that a spheroidal model be used. Such models are essential, for example, when you are mapping high-resolution satellite or aerial imagery, or when you are working with coordinates from the Global Positioning System (GPS). This section addresses how Mapping Toolbox software accurately models the shape, or figure, of the Earth.

Ellipsoid Shape

You can define ellipsoids in several ways. They are usually specified by a semimajor and a semiminor axis, but are often expressed in terms of a semimajor axis and either inverse flattening (which for the Earth, as mentioned above, is one part in 300) or eccentricity. Whichever parameters are used, as long as an axis length is included, the ellipsoid is fully constrained and the other parameters are derivable. The components of an ellipsoid are shown in the following diagram.



The toolbox includes ellipsoid models that represent the figures of the Sun, Moon, and planets, as well as a set of the most common ellipsoid models of the Earth. For more information, see “Comparison of Reference Spheroids” on page 8-4.

Geoid Shape

Literally, *geoid* means *Earth-shaped*. The geoid is an empirical approximation of the figure of the Earth (minus topographic relief), its "lumpiness." Specifically, it is an equipotential surface with respect to gravity, more or less corresponding to mean sea level. It is approximately an ellipsoid, but not exactly so because local variations in gravity create minor hills and dales (which range from -100 m to +60 m across the Earth). This variation in height is on the order of 1 percent of the differences between the semimajor and semiminor ellipsoid axes used to approximate the Earth's shape.

The shape of the geoid is important for some purposes, such as calculating satellite orbits, but need not be taken into account for every mapping application. However, knowledge of the geoid is

sometimes necessary, for example, when you compare elevations given as height above mean sea level to elevations derived from GPS measurements. Geoid representations are also inherent in datum definitions.

Map the Geoid

Get geoid heights and a geographic postings reference object from the EGM96 geoid model. Load coastline latitude and longitude data.

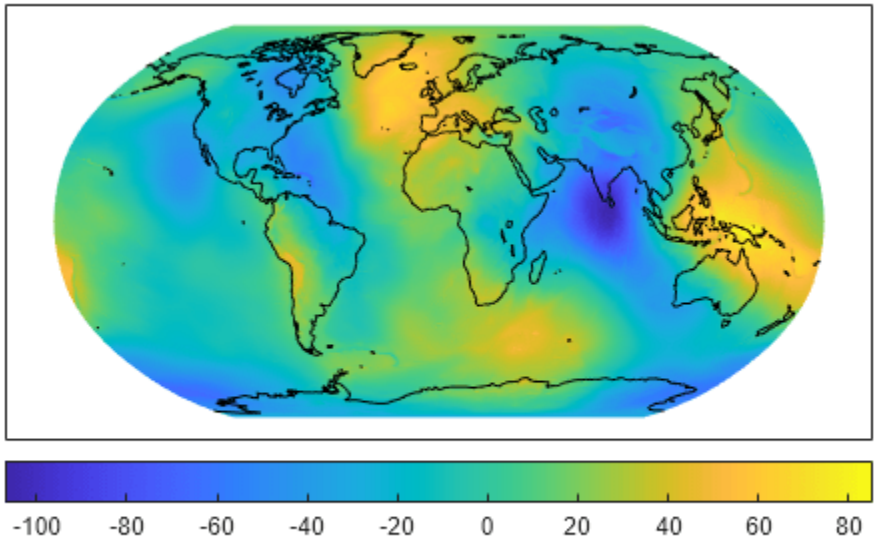
```
[N,R] = egm96geoid;
load coastlines
```

Display the geoid heights as a surface using a Robinson projection. Ensure the coastline data appears over the surface by setting the `'CData'` name-value pair to the geoid heights data and the `'ZData'` name-value pair to a matrix of zeros. Then, display the coastline data.

```
axesm robinson
Z = zeros(R.RasterSize);
geoshow(N,R,'DisplayType','surface','CData',N,'ZData',Z)
geoshow(coastlat,coastlon,'color','k')
```

Display a colorbar below the map.

```
colorbar('southoutside')
```



Comparison of Reference Spheroids

When the Earth (or another roughly spherical body such as the Moon) is modeled as a sphere having a standard radius, it is called a *reference sphere*. Likewise, when the model is a flattened (oblate) ellipsoid of revolution, with a standard semimajor axis and standard inverse flattening, semiminor axis, or eccentricity, it is called a *reference ellipsoid*. Both models are spheroidal in shape, so each can be considered to be a type of *reference spheroid*. Mapping Toolbox supports several representations for reference spheroids: `referenceSphere`, `referenceEllipsoid`, and `oblateSpheroid` objects, and an older representation, *ellipsoid vector*.

referenceSphere Objects

When using a strictly spherical model, you should generally use a `referenceSphere` object (although both `referenceEllipsoid` and `oblateSpheroid` can represent a perfect sphere).

By default, `referenceSphere` returns a dimensionless unit sphere:

```
referenceSphere
```

```
ans =

referenceSphere with defining properties:

          Name: 'Unit Sphere'
    LengthUnit: ''
        Radius: 1

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

You can request a specific body by name, and the radius will be in meters by default:

```
earth = referenceSphere('Earth')
```

```
earth =

referenceSphere with defining properties:

          Name: 'Earth'
    LengthUnit: 'meter'
        Radius: 6371000

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
```

```
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

You can reset the length unit if desired (and the radius is rescaled appropriately) :

```
earth.LengthUnit = 'kilometer'

earth =

referenceSphere with defining properties:

          Name: 'Earth'
    LengthUnit: 'kilometer'
        Radius: 6371

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

or specify the length unit at the time of construction:

```
referenceSphere('Earth','km')

ans =

referenceSphere with defining properties:

          Name: 'Earth'
    LengthUnit: 'kilometer'
        Radius: 6371

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Any length unit supported by `validateLengthUnit` can be used. A variety of abbreviations are supported for most length units, see `validateLengthUnit` for a complete list.

One thing to note about `referenceSphere` is that only the defining properties are displayed, in order to reduce clutter at the command line. (This approach saves a small amount of computation as well.) In particular, don't overlook the dependent `SurfaceArea` and `Volume` properties, even though they are not displayed. The surface area of the spherical earth model, for example, is easily obtained through the `SurfaceArea` property:

```
earth.SurfaceArea
```

```
ans =
   5.1006e+08
```

This result is in square kilometers, because the `LengthUnit` property of the object earth has value `'kilometer'`.

When programming with Mapping Toolbox it may help to be aware that `referenceSphere` actually includes all the geometric properties of `referenceEllipsoid`and `oblateSpheroid` (`SemimajorAxis`, `SemiminorAxis`, `InverseFlattening`, `Eccentricity`, `Flattening`, `ThirdFlattening`, and `MeanRadius`, as well as `SurfaceArea`, and `Volume`). None of these properties can be set on a `referenceSphere`, and some have values that are fixed for all spheres. `Eccentricity` is always `0`, for example. But they provide a flexible environment for programming because any geometric computation that accepts a `referenceEllipsoid` will also run properly given a `referenceSphere`. This is a type of polymorphism in which different classes support common, or strongly overlapping interfaces.

referenceEllipsoid Objects

When using an oblate spheroid to represent the Earth (or another roughly spherical body), you should generally use a `referenceEllipsoid` object. An important exception occurs with certain small-scale map projections, many of which are defined only on the sphere. However, all important projections used for large-scale work, including Transverse Mercator and Lambert Conformal Conic, are defined on the ellipsoid as well as the sphere.

Like `referenceSphere`, `referenceEllipsoid` returns a dimensionless unit sphere by default:

```
referenceEllipsoid
```

```
ans =

referenceEllipsoid with defining properties:

                Code: []
                Name: 'Unit Sphere'
          LengthUnit: ''
        SemimajorAxis: 1
        SemiminorAxis: 1
    InverseFlattening: Inf
         Eccentricity: 0

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

More typically, you would request a specific ellipsoid by name, resulting in an object with semimajor and semiminor axes properties in meters. For example, the following returns a `referenceEllipsoid` with `SemimajorAxis` and `InverseFlattening` property settings that match the defining parameters of Geodetic Reference System 1980 (GRS 80).

```
grs80 = referenceEllipsoid('Geodetic Reference System 1980')
```

```
grs80 =

referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'Geodetic Reference System 1980'
          LengthUnit: 'meter'
        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

In general, you should use the reference ellipsoid corresponding to the geodetic datum to which the coordinates of your data are referenced. For instance, the GRS 80 ellipsoid is specified for use with coordinates referenced to the North American Datum of 1983 (NAD 83).

As in the case of `referenceSphere`, you can reset the length unit if desired:

```
grs80.LengthUnit = 'kilometer'
```

```
grs80 =

referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'Geodetic Reference System 1980'
          LengthUnit: 'kilometer'
        SemimajorAxis: 6378.137
        SemiminorAxis: 6356.75231414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

or specify the length unit at the time of construction:

```
referenceEllipsoid('Geodetic Reference System 1980','km')
```

```
ans =

referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'Geodetic Reference System 1980'
          LengthUnit: 'kilometer'
        SemimajorAxis: 6378.137
        SemiminorAxis: 6356.75231414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Any length unit supported by `validateLengthUnit` can be used.

The command-line display includes four geometric properties: `SemimajorAxis`, `SemiminorAxis`, `InverseFlattening`, and `Eccentricity`. Any pair of these properties, as long as at least one is an axis length, is sufficient to fully define a oblate spheroid; the four properties constitute a mutually dependent set. Parameters `InverseFlattening` and `Eccentricity` as a set are not sufficient to define an ellipsoid because both are dimensionless shape properties. Neither of those parameters provides a length scale, and, furthermore, are mutually dependent: `ecc = sqrt((2 - f) * f)`.

In addition, there are five dependent properties that are not displayed, in order to reduce clutter on the command line: `Flattening`, `ThirdFlattening`, `MeanRadius`, `SurfaceArea`, and `Volume`. `SurfaceArea` and `Volume` work the same way as their `referenceSphere` counterparts. To continue the preceding example, the surface area of the GRS 80 ellipsoid in square kilometers (because `LengthUnit` is `'kilometer'`), is easily obtained as follows:

```
grs80.SurfaceArea

ans =
   5.1007e+08
```

See the `referenceEllipsoid` reference page for definitions of the shape properties, permissible values for the `Name` property, and information on the `Code` property.

World Geodetic System 1984

Due in part to widespread use of the U.S. NAVSTAR Global Positioning System (GPS), which is tied to World Geodetic System 1984 (WGS 84), the WGS 84 reference ellipsoid is often the appropriate choice. For both convenience and speed (obtained by bypassing a table look-up step), it's a good idea in this case to use the `wgs84Ellipsoid` function, for example,

```
wgs84 = wgs84Ellipsoid;
```

The preceding line is equivalent to:

```
wgs84 = referenceEllipsoid('wgs84');
```

but it is easier to type and faster to run. You can also specify a length unit. `wgs84Ellipsoid(lengthUnit)`, is equivalent to `referenceEllipsoid('wgs84',lengthUnit)`, where `lengthUnit` is any unit value accepted by the `validateLengthUnit` function.

For example, the following two commands show that the surface area of the WGS 84 ellipsoid is a little over 5 x 10^14 square meters:

```
s = wgs84Ellipsoid

s =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
          LengthUnit: 'meter'
        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31424518
    InverseFlattening: 298.257223563
         Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume

s.SurfaceArea

ans =

   5.1007e+14
```

Ellipsoid Vectors

An ellipsoid vector is simply a 2-by-1 double of the form: `[semimajor_axis eccentricity]`. Unlike a spheroid object (any instance of `referenceSphere`, `referenceEllipsoid`, or `oblateSpheroid`), an ellipsoid vector is not self-documenting. Ellipsoid vectors are not even self-identifying. You have to know that a given 2-by-1 vector is indeed an ellipsoid vector to make any use of it. This representation does not validate that `semimajor_axis` is real and positive, for example, you have to do such validations for yourself.

Many toolbox functions accept ellipsoid vectors as input, but such functions accept spheroid objects as well and, for the reasons just stated, spheroid objects are recommended over ellipsoid vectors. In case you have written a function of your own that requires an ellipsoid vector as input, or have received such a function from someone else, note that you can easily convert any spheroid object `s` into an ellipsoid vector as follows:

```
[s.SemimajorAxis s.Eccentricity]
```

This means that you can construct a spheroid object using any of the three class constructors, or the `wgs84Ellipsoid` function, and hand off the result in the form of an ellipsoid vector if necessary.

oblateSpheroid Objects

`oblateSpheroid` is the superclass of `referenceEllipsoid`. An `oblateSpheroid` object is just like a `referenceEllipsoid` object minus its `Code`, `Name`, and `LengthUnit` properties. In fact, the primary role of the `oblateSpheroid` class is to provide the purely geometric properties and behaviors needed by the `referenceEllipsoid` class.

For most purposes, you can simply ignore this distinction, and the `oblateSpheroid` class itself, as a matter of internal software composition. No harm will come about, because a `referenceEllipsoid` object can do anything and be used anywhere that an `oblateSpheroid` can.

However, you can use `oblateSpheroid` directly when dealing with an ellipsoid vector that lacks a specified name or length unit. For example, compute the volume of an ellipsoid with a semimajor axis of 2000 and eccentricity of 0.1, as shown in the following.

```
e = [2000 0.1];
s = oblateSpheroid;
s.SemimajorAxis = e(1);
s.Eccentricity = e(2)
s.Volume
```

```
s =

oblateSpheroid with defining properties:

        SemimajorAxis: 2000
        SemiminorAxis: 1989.97487421324
    InverseFlattening: 199.498743710662
         Eccentricity: 0.1

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume


ans =

   3.3342e+10
```

Of course, since the length unit of `e` is unspecified, the unit of `s.Volume` is likewise unspecified.

Work with Reference Spheroids

Reference spheroids are needed in three main contexts: map projections, curves and areas on the surface of a spheroid, and 3-D computations involving geodetic coordinates.

Map Projections

You can set the value of the `Geoid` property of a new `axesm`-based map (which is actually a Spheroid property) using any type of reference spheroid representation. Except in the case of UTM and UPS, the default value is an ellipsoid vector representing the unit sphere: `[1 0]`. It is also the default value when using the `worldmap` and `usamap` functions.

You can reset the `Geoid` property of an existing `axesm`-based map to any type of reference spheroid representation by using `setm`. For example, `worldmap` always sets up a projection based on the unit sphere but you can subsequently use `setm` to switch to the spheroid of your choice. To set up a map of North America for use with Geodetic Reference System 1980, for instance, follow `worldmap` with a call to `setm`, like this:

```
ax = worldmap('North America');
setm(ax,'geoid',referenceEllipsoid('grs80'))
```

When projecting or unprojecting data without an `axesm`-based map, you can set the `geoid` field of a map projection structure (`mstruct`) to any type of reference spheroid representation. Remember to follow all `mstruct` updates with a second call to `defaultm` to ensure that all properties are set to legitimate values. For example, to use the Miller projection with WGS 84 in kilometers, start with:

```
mstruct = defaultm('miller');
mstruct.geoid = wgs84Ellipsoid('km');
mstruct = defaultm(mstruct);
```

You can inspect the `mstruct` to ensure that you are indeed using the WGS 84 ellipsoid:

```
mstruct.geoid
```

```
ans =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
          LengthUnit: 'kilometer'
        SemimajorAxis: 6378.137
        SemiminorAxis: 6356.75231424518
    InverseFlattening: 298.257223563
         Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

See axesm-Based Map Properties for definitions of the fields found in `mstructs`.

Curves and Areas

Another important context in which reference spheroids appear is the computation of curves and areas on the surface of a sphere or oblate spheroid. The `distance` function, for example, assumes a sphere by default, but accepts a reference spheroid as an optional input. `distance` is used to compute the length of the geodesic or rhumb line arc between a pair of points with given latitudes and longitudes. If a reference spheroid is provided through the `ellipsoid` argument, then the unit used for the arc length output matches the `LengthUnit` property of the spheroid.

Other functions for working with curves and areas that accept reference spheroids include `reckon`, `scircle1`, `scircle2`, `ellipse1`, `track1`, `track2`, and `areaquad`, to name just a few. When using such functions without their `ellipsoid` argument, be sure to check the individual function help if you are unsure about which reference spheroid is assumed by default.

3-D Coordinate Transformations

The third context in which reference spheroids frequently appear is the transformation of geodetic coordinates (latitude, longitude, and height above the ellipsoid) to other coordinate systems. For example, the `geodetic2ecef` function, which converts point locations from a geodetic system to a geocentric (Earth-Centered Earth-Fixed) Cartesian system, requires a reference spheroid object (or an ellipsoid vector) as input.

Comparison of 3-D Coordinate Systems

Coordinate systems represent position on the Earth using coordinates. Mapping Toolbox functions transform coordinates between Earth-centered Earth-fixed (ECEF), geodetic, east-north-up (ENU), north-east-down (NED), and azimuth-elevation-range (AER) systems.

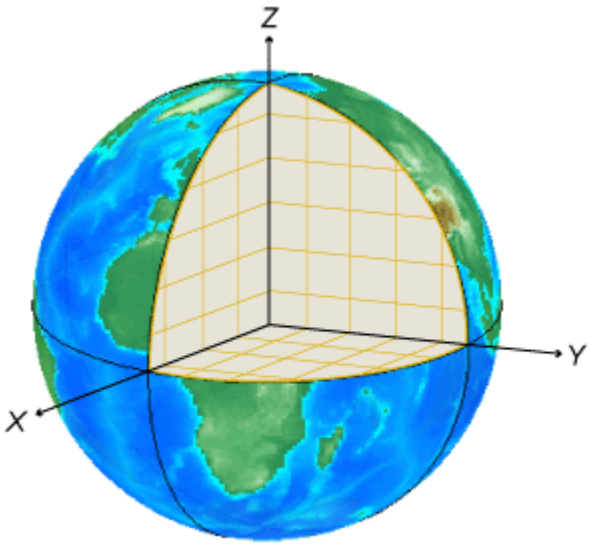
Global systems such as ECEF and geodetic systems describe the position of an object using a triplet of coordinates. Local systems such as ENU, NED, and AER systems require two triplets of coordinates: one triplet describes the location of the origin, and the other triplet describes the location of the object with respect to the origin.

When you work with 3-D coordinate systems, you must specify an ellipsoid model that approximates the shape of the Earth. For more information about ellipsoid models, see “Comparison of Reference Spheroids” on page 8-4. All of the sample coordinates on this page use the World Geodetic System of 1984 (WGS84).

Earth-Centered Earth-Fixed Coordinates

An Earth-centered Earth-fixed (ECEF) system uses the Cartesian coordinates (X,Y,Z) to represent position relative to the center of the reference ellipsoid. The distance between the center of the ellipsoid and the center of the Earth depends on the reference ellipsoid.

- The positive X-axis intersects the surface of the ellipsoid at 0° latitude and 0° longitude, where the equator meets the prime meridian.
- The positive Y-axis intersects the surface of the ellipsoid at 0° latitude and 90° longitude.
- The positive Z-axis intersects the surface of the ellipsoid at 90° latitude and 0° longitude, the North Pole.

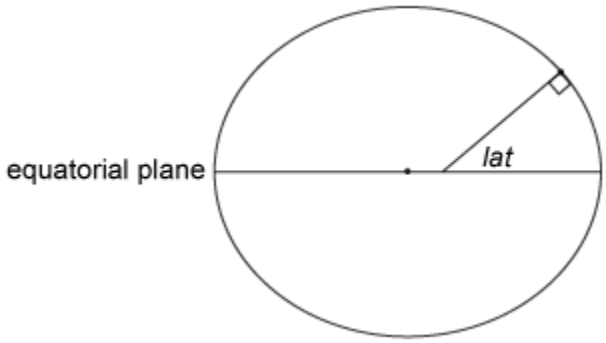


For example, the ECEF coordinates of Parc des Buttes-Chaumont are (4198945 m, 174747 m, 4781887 m).

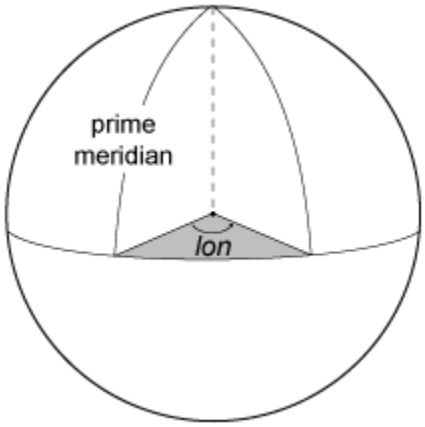
Geodetic Coordinates

A geodetic system uses the coordinates (*lat*,*lon*,*h*) to represent position relative to a reference ellipsoid.

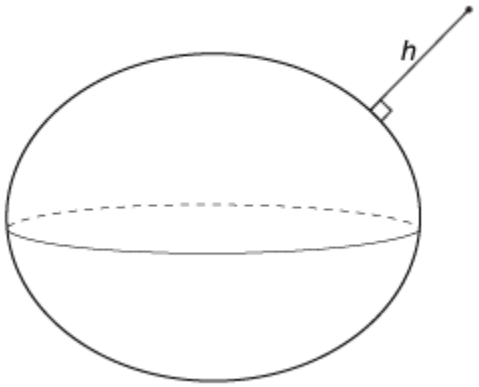
- *lat*, the latitude, originates at the equator. More specifically, the latitude of a point is the angle that a normal vector to the ellipsoid at that point makes with the equatorial plane, which contains the center and equator of the ellipsoid. An angle of latitude is within the range [–90°, 90°]. Positive latitudes correspond to north and negative latitudes correspond to south.



- *lon*, the longitude, originates at the prime meridian. More specifically, the longitude of a point is the angle that a plane containing the ellipsoid center and the meridian containing that point makes with the plane containing the ellipsoid center and prime meridian. Positive longitudes are measured in a counterclockwise direction from a vantage point above the North Pole. Typically, longitude is within the range [–180°, 180°] or [0°, 360°].



- *h*, the ellipsoidal height, is measured along a normal of the reference spheroid. Coordinate transformation functions such as `geodetic2ecef` require you to specify *h* in the same units as the reference ellipsoid. You can change the units of the reference ellipsoid using the `LengthUnit` property. Terrain models typically supply data using orthometric height rather than ellipsoidal height. For information about calculating ellipsoidal height from orthometric height, see “Find Ellipsoidal Height from Orthometric and Geoid Height” on page 8-28.

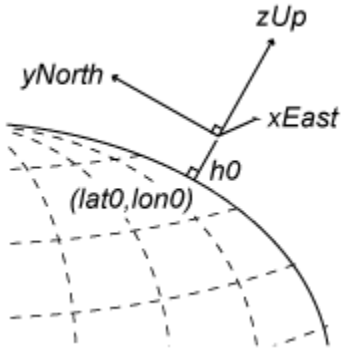


For example, the geodetic coordinates of Parc des Buttes-Chaumont are (48.8800°, 2.3831°, 124.5089 m).

East-North-Up Coordinates

An east-north-up (ENU) system uses the Cartesian coordinates (*xEast*,*yNorth*,*zUp*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Note that the origin does not necessarily lie on the surface of the ellipsoid.

- The positive *xEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *yNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *zUp*-axis points upward along the ellipsoid normal.

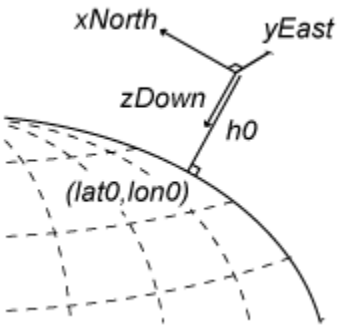


For example, Montmartre has geodetic coordinates (48.8862°, 2.3343°, 174.5217 m). The ENU coordinates of Parc des Buttes-Chaumont with respect to Montmartre are (3579.4232 m, –688.3514 m, –51.0524 m).

North-East-Down Coordinates

A north-east-down (NED) system uses the Cartesian coordinates (*xNorth*,*yEast*,*zDown*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Typically, the local origin of an NED system is above the surface of the Earth.

- The positive *xNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *yEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *zDown*-axis points downward along the ellipsoid normal.



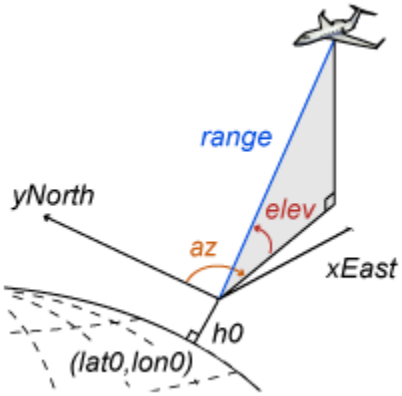
An NED coordinate system is commonly used to specify location relative to a moving aircraft. In this application, the origin and axes of an NED system change continuously. Note that the coordinates are not fixed to the frame of the aircraft.

For example, an aircraft flying into Charles de Gaulle airport has geodetic coordinates (48.9978°, 2.7594°, 699.8683 m). The NED coordinates of the airport with respect to the plane are (1645.8313 m, –15677.1868 m, 555.8221 m).

Azimuth-Elevation-Range Coordinates

An azimuth-elevation-range (AER) system uses the spherical coordinates (*az*,*elev*,*range*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Azimuth, elevation, and slant range are dependent on a local Cartesian system, for example, an ENU system.

- *az*, the azimuth, is the clockwise angle in the *xEast*-*yNorth* plane from the positive *yNorth*-axis to the projection of the object into the plane.
- *elev*, the elevation, is the angle from the *xEast*-*yNorth* plane to the object.
- *range*, the slant range, is the Euclidean distance between the object and the local origin.



For example, a lidar sensor at the Charles de Gaulle airport has geodetic coordinates (48.0124°, 2.5451°, 163.4885 m). The AER coordinates of an airplane with respect to the sensor are (95.8314°, 1.8781°, 15773.1381 m).

Tips

If you are transforming coordinates between ENU, NED, and AER systems with the same origin, then you do not need to specify a reference ellipsoid or the coordinates of the origin.

See Also

`geodetic2ecef` | `ecef2enu` | `enu2aer` | `aer2ned` | `ned2geodetic` | `geodetic2aer`

More About

- “Comparison of Reference Spheroids” on page 8-4

References

[1] Guowei, C., B.M. Cheh, and T. H. Lee. *Unmanned Rotorcraft Systems*. London: Springer-Verlag London Limited: 2011.

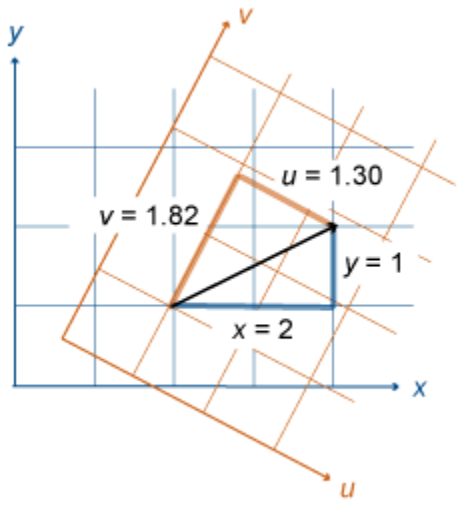
[2] Van Sickle, J. *Basic GIS Coordinates*. Boca Raton, FL: CRC Press LLC, 2004.

Vectors in 3-D Coordinate Systems

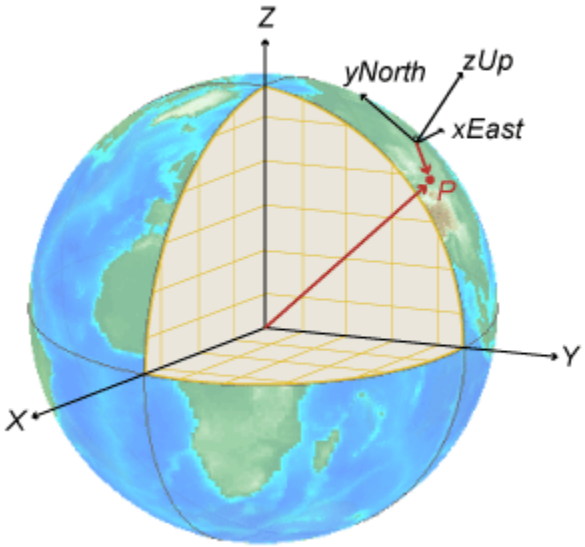
Vectors represent quantities such as velocity and acceleration. Mapping Toolbox functions transform vector components between Earth-centered Earth-fixed (ECEF) and east-north-up (ENU) or north-east-down (NED) systems. For more information about ECEF, ENU, and NED coordinate systems, see “Comparison of 3-D Coordinate Systems” on page 8-13.

Unlike coordinates that measure position, vector components in a Cartesian system do not depend on a position in space. Therefore, when you transform a vector from one system to another, only the components of the vector change. The magnitude of the vector remains the same.

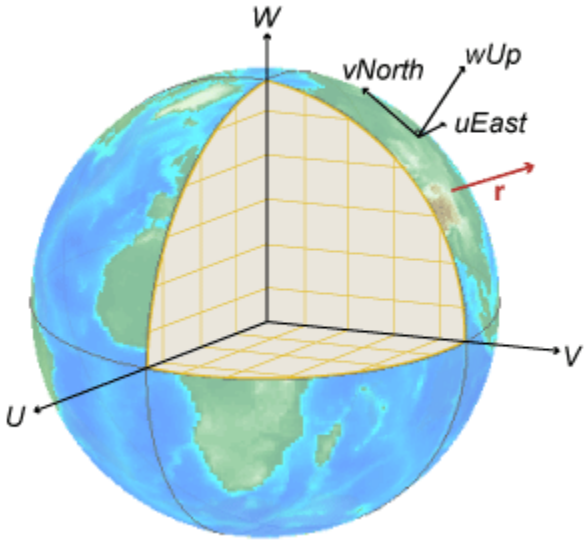
For example, this image shows a 2-D vector transformation from an *x*-*y* system to a *u*-*v* system. The vector has components $x = 2$ and $y = 1$ in the *x*-*y* system, and components $u = 1.30$ and $v = 1.82$ in the *u*-*v* system. The components of the vector are different, but in each system the magnitude of the vector is 2.24 units.



This image shows a coordinate transformation from a global ECEF system to a local ENU system using `ecef2enu`. The position vectors start at the origin of each system and end at point *P*. Therefore, the transformation changes the magnitude of the position vector.



This image shows a vector transformation from a global ECEF system to a local ENU system using `ecef2enuv`. The vector **r** does not depend on a position. Therefore, the transformation changes the components of the vector, but the magnitude of the vector is the same.



Tips

Unlike coordinate transformation functions such as `ecef2enu`, vector transformation functions such as `ecef2enuv` do not require you to specify a reference spheroid or the ellipsoidal height of the local origin. The geodetic latitude and longitude of the local origin is sufficient to define the orientation of the *uEast*, *vNorth*, and *wUp* axes.

See Also

`enu2ecefv` | `ned2ecefv` | `ecef2nedv` | `ecef2enuv`

More About

- “Comparison of 3-D Coordinate Systems” on page 8-13

Plotting a 3-D Dome as a Mesh over a Globe

This example shows how to start with a 3-D feature in a system of local east-north-up (ENU) coordinates, then transform and combine it with a globe display in Earth-Centered, Earth-Fixed (ECEF) coordinates.

Step 1: Set Defining Parameters

Use Geodetic Reference System 1980 (GRS80) and work in units of kilometers. Place the origin of the local system near Washington, DC, USA.

```
grs80 = referenceEllipsoid("grs80","km");
domeRadius =  3000;  % km
domeLat =  39;       % degrees
domeLon = -77;       % degrees
domeAlt = 0;         % km
```

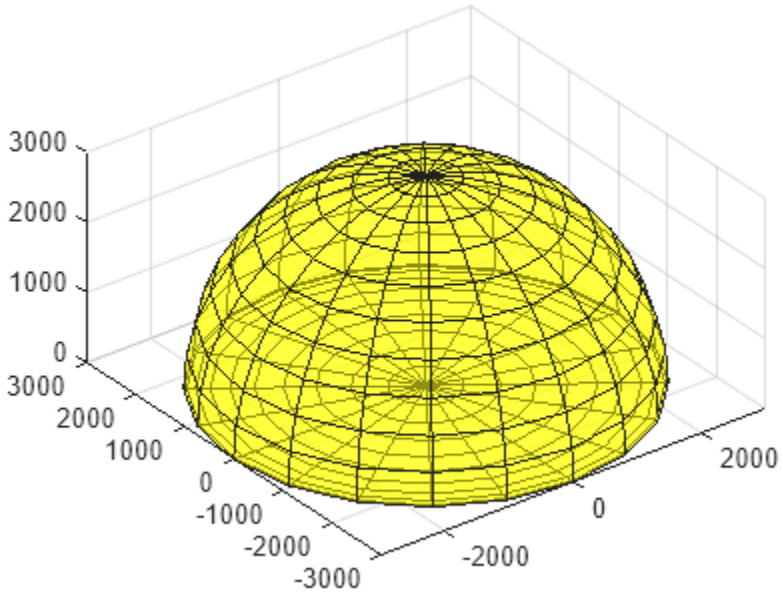
Step 2: Construct the Dome in Local East-North-Up Coordinates

The local ENU system is defined with respect to a geodetic reference point, specified in this case by (`domeLat`, `domeLon`, and `domeAlt`). It is a 3-D Cartesian system in which the positive x-axis is directed to the east, the positive y-axis is directed to the north, and the z-axis is normal to the reference ellipsoid and directed upward.

In this example, the 3-D feature is a hemisphere in the z >= 0 half-space with a radius of 3000 kilometers. This hemisphere could enclose, hypothetically, the volume of space within range of a idealized radar system having uniform coverage from the horizon to the zenith, in all azimuths. Volumes of space such as this, when representing zones of effective surveillance coverage, are sometimes known informally as "radar domes."

A quick way to construct coordinate arrays outlining a closed hemispheric dome is to start with a unit sphere, scale up the radius, and collapse the lower hemisphere. It's easier to visualize if you make it semitransparent -- setting the `FaceAlpha` to 0.5 in this case.

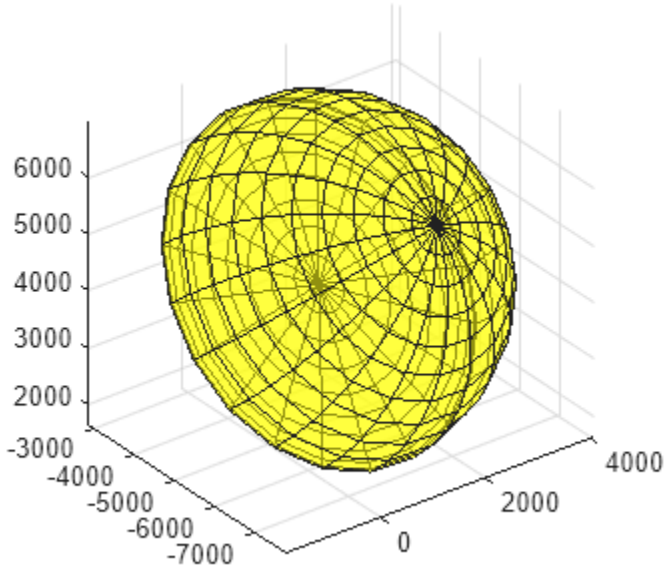
```
[x,y,z] = sphere(20);
xEast  = domeRadius * x;
yNorth = domeRadius * y;
zUp    = domeRadius * z;
zUp(zUp < 0) = 0;
surf(xEast,yNorth,zUp,"FaceColor","yellow","FaceAlpha",0.5)
axis equal
```



Step 3: Convert Dome to the Earth-Centered Earth-Fixed (ECEF) System

Use the `enu2ecef` function to convert the dome from local ENU to an ECEF system, based on the GRS 80 reference ellipsoid. It applies a 3-D translation and rotation. Notice how the hemisphere becomes tilted and how its center moves thousands of kilometers from the origin.

```
[xECEF, yECEF, zECEF] ...
    = enu2ecef(xEast,yNorth,zUp,domeLat,domeLon,domeAlt,grs80);
surf(xECEF,yECEF,zECEF,"FaceColor","yellow","FaceAlpha",0.5)
axis equal
```



Step 4: Construct a Globe Display

Construct a basic globe display using `axesm` and `globe`.

```
figure
ax = axesm("globe","Geoid",grs80,"Grid","on", ...
    "GLineWidth",1,"GLineStyle","-",...
    "Gcolor",[0.9 0.9 0.1],"Galtitude",100);
ax.Position = [0 0 1 1];
axis equal off
view(3)
```

Step 5: Add Various Global Map Data

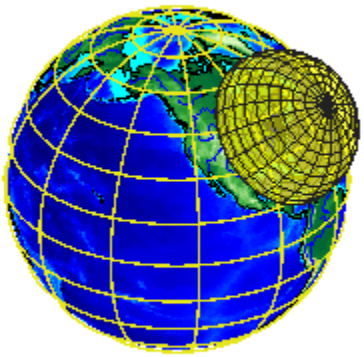
Add low-resolution global topography, coastlines, and rivers to the globe.

```
load topo60c
geoshow(topo60c,topo60cR,"DisplayType","texturemap")
demcmap(topo60c)

load coastlines
geoshow(coastlat,coastlon,"Color","black")

rivers = readgeotable("worldrivers.shp");
geoshow(rivers,"Color","blue")
```

Step 6: Add the Dome to the Globe Display

Add the ECEF version of dome to the globe axes as a semitransparent mesh.

```
surf(xECEF,yECEF,zECEF,"FaceColor","yellow","FaceAlpha",0.5)
```



You can view the dome and globe from different angles by interactively rotating the axes.

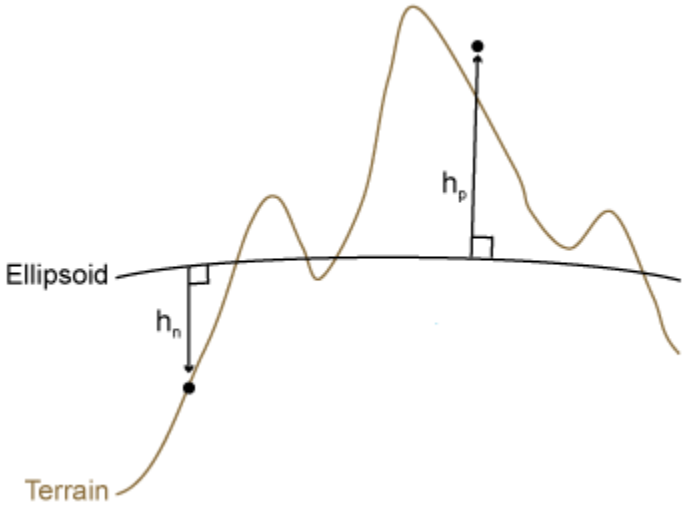
Credit

Thanks to Edward J. Mayhew, Jr. for providing technical background on "radar domes" and for bringing to our attention the problem of visualizing them with the Mapping Toolbox™.

Find Ellipsoidal Height from Orthometric Height

The *height* of an object may refer to its ellipsoidal height or its orthometric height. Mapping Toolbox functions such as `geodetic2enu` require the input argument *ellipsoidal height*, but data often quantifies *orthometric height* instead. You can convert orthometric height to ellipsoidal height by using a geoid model.

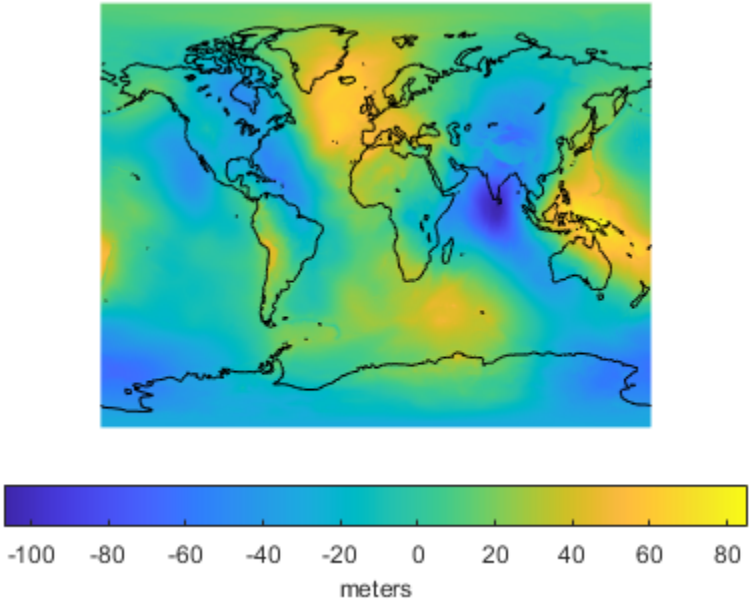
Ellipsoidal height, called h, is height measured along a normal of a reference ellipsoid. For more information about reference ellipsoids, see “Comparison of Reference Spheroids” on page 8-4. This image shows a positive ellipsoidal height, h_p, and a negative ellipsoidal height, h_n.



Most terrain models provide data using orthometric height instead of ellipsoidal height. Orthometric height, called H, is height above the geoid.

The *geoid* models the average sea level of the Earth without effects such as weather, tides, and land. A geoid model is created by measuring variations in the Earth's gravitational field, so it has a smoothly undulating shape. Orthometric height is measured relative to the geoid.

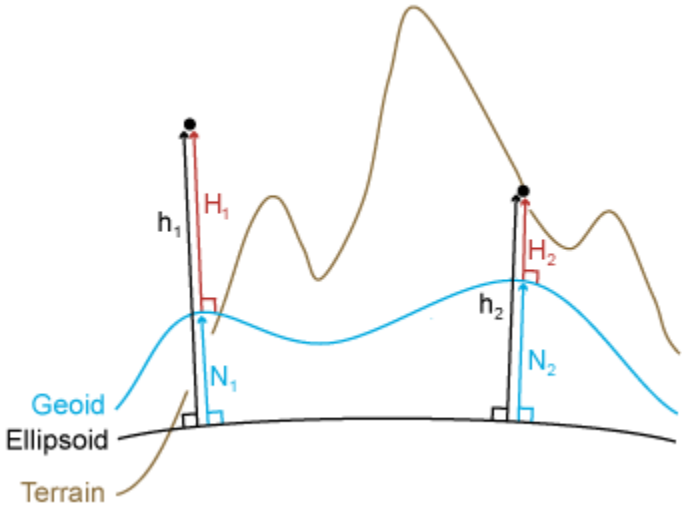
Geoid height, called N, is the height of the geoid measured along a normal of a reference ellipsoid. For example, geoid height values from the Earth Gravitational Model of 1996 (EGM96) are referenced to the ellipsoid defined by the World Geodetic System of 1984 (WGS84). Below is an illustration of the geoid from EGM96, with geoid heights in meters.



To find ellipsoidal height at a specified latitude and longitude, add the orthometric height and geoid height: $h = H + N$. You can find the height of the geoid from EGM96 at specified latitudes and longitudes using the `egm96geoid` function.

The equation $h = H + N$ is an approximation because the direction along which a geoid height is measured is not necessarily the same as the direction along which an orthometric height is measured. However, the approximation is suitable for most practical purposes.

The following image illustrates the relationship between ellipsoidal height, orthometric height, and the geoid. The values h_1, H_1, and N_1 demonstrate the relationship for an airborne object, while h_2, H_2, and N_2 demonstrate the relationship for an object on land.



Find Ellipsoidal Height from Orthometric and Geoid Height

Find the ellipsoidal height of the summit of Mount Everest, using its orthometric height and a geoid model.

Specify the latitude and longitude of the summit in degrees. Specify the orthometric height in meters.

```
lat = 27.988056;
lon = 86.925278;
H = 8848;
```

Find the height of the geoid at the location specified by `lat` and `lon` using `egm96geoid`.

```
N = egm96geoid(lat,lon);
```

Calculate the ellipsoidal height of the summit.

```
h = H + N
```

```
h =
8.8193e+03
```

References

[1] NOAA. "What is the geoid?" *National Ocean Service website*. https://oceanservice.noaa.gov/facts/geoid.html, 06/25/18.

See Also

`egm96geoid` | `geodetic2enu`

More About

- “Comparison of Reference Spheroids” on page 8-4

Angle Representations and Angular Units

In this section...
“Degrees and Radians” on page 8-30
“Degree Representations” on page 8-30
“Latitude and Longitude Formats” on page 8-31

This topic describes the angular units that Mapping Toolbox functions use, how to convert between common angle representations, and how to format latitude and longitude angles as text.

Use angles in mapping applications to indicate absolute positions, relative positions along reference ellipsoids, and relative positions in 3-D:

- Absolute positions are latitudes and longitudes.
- Relative positions on reference ellipsoids are azimuths along geodesics, great circles, and rhumb lines. Find these relative positions by using the `distance` function.
- Relative positions in 3-D are both azimuths and elevations. Find these relative positions by using the `geodetic2aer` function.

Degrees and Radians

The most common angular units are degrees and radians. Many Mapping Toolbox functions perform angle computations in degrees. If your data is in radians, you can convert to degrees by using the `rad2deg` function.

Many Mapping Toolbox functions, such as `distance` and `azimuth`, use degrees by default and allow you to choose radians. Some functions, such as `unwrapMultipart` and `meridianarc`, use radians by default or require you to work in radians.

Degree Representations

Angles are commonly represented using degrees (–35.2625°), degrees-minutes (–35° 15.75'), and degrees-minutes-seconds (–35° 15' 45"). Minutes are 1/60 of a degree and seconds are 1/60 of a minute.

Because Mapping Toolbox functions perform angle computations using only degrees, if your data has values in degrees-minutes (DM) or degrees-minutes-seconds (DMS), you must convert the values to degrees before using them as input. Convert numeric values from DM or DMS to degrees by using the `dm2degrees` or `dms2degrees` function. You can also convert text values from DM or DMS to degrees by using the `str2angle` function.

If you want to publish coordinate values or format data for use with other applications, then you can convert degrees to DM or DMS. Convert degrees to DM or DMS by using the `degrees2dm` or `degrees2dms` function.

Degrees

Degrees represent angles using a sign or direction and a nonnegative decimal number. For example, you can represent a longitude of 35.2625 degrees west of the prime meridian as –35.2625° or 35.2625° W.

Note Decimal degrees is a common way to refer to noninteger latitude and longitude values. The term is appropriate for angles formatted in text using decimal notation, such as when printed in a document or displayed at the MATLAB command line. However, the term is inaccurate for angles in degrees that are stored in computer memory, such as in MATLAB variables. Angles are stored in memory as single or double precision floating-point numbers, which are binary representations, not decimal representations. Therefore, the term *degrees* is more accurate than decimal degrees for angles stored in memory, even when the angles have noninteger values.

Degrees-Minutes

Degrees-minutes represent angles using a sign or direction and two numbers:

- Degrees (°) — A signed or unsigned integer
- Minutes (') — A nonnegative decimal number in the range [0 60)

For example, a longitude of –35.2625 in degrees is –35° 15.75' or 35° 15.75' W in DM. This code shows how to convert numbers in DM to degrees by using the `dm2degrees` function.

```
dm2degrees([-35 15.75])
```

```
ans =

  -35.2625
```

Degrees-Minutes-Seconds

Degrees-minutes-seconds represent angles using a sign or direction and three numbers:

- Degrees (°) — A signed or unsigned integer
- Minutes (') — A nonnegative integer in the range [0 59]
- Seconds (") — A nonnegative decimal number in the range [0 60)

For example, a longitude of –35.2625 in degrees is –35° 15' 45" or 35° 15' 45" W in DMS. This code shows how to convert numbers in DMS to degrees by using the `dms2degrees` function.

```
dms2degrees([-35 15 45])
```

```
ans =

  -35.2625
```

Latitude and Longitude Formats

You can format latitudes and longitudes by using letters or symbols:

- Degrees — d or °
- Minutes — m or '
- Seconds — s or "

You can also indicate the sign of the angle by using letters:

- Positive latitude — N

- Negative latitude — S
- Positive longitude — E
- Negative longitude — W

For example, you can format 35 degrees, 15 minutes, 45 seconds west of the prime meridian as 35d15m45sW, 35° 15' 45" W, or –35° 15' 45".

Convert latitudes and longitudes in degrees to character arrays in DM or DMS for use with LaTeX by using the `angl2str` function. This code shows how to convert a longitude of `-36.2625` degrees to a character array that uses DMS.

```
angl2str(-35.2625,"ew","degrees2dms")
```

```
ans =

    ' 35^{\circ} 15' 45.00" W '
```

If you want to format angles in ways that the `angl2str` function does not support, first convert the angle from decimal degrees to numbers in DM or DMS by using the `degrees2dms` or `degrees2dm` function. Then, format the numbers into a string or character vector by using the `sprintf` function.

This code shows how to convert the same longitude to a character array that uses Unicode® instead of LaTeX.

```
dm = degrees2dms(-35.2625);
sprintf('%d\x00B0 %u\x0027 %.2f\x0022 W',dm)
```

```
ans =

    '-35° 15' 45.00" W'
```

See Also

Functions
`toDegrees`

Length and Distance Units

Linear measurements of lengths and distances on spheres and spheroids can use the same units they do on the plane, such as feet, meters, miles, and kilometers. They can be used for

- Absolute positions, such as map coordinates or terrain elevations
- Dimensions, such as a planet's radius or its semimajor and semiminor axes
- Distances between points or along routes, in 2-D or 3-D space or across terrain

Length units are needed to describe

- The dimensions of a reference sphere or ellipsoid
- The line-of-sight distance between points
- Distances along great circle or rhumb line curves on an ellipsoid or sphere
- *X-Y* locations in a projected coordinate system or map grid
- Offsets from a map origin (false eastings and northings)
- *X-Y-Z* locations in Earth-centered Earth-fixed (ECEF) or local vertical systems
- Heights of various types (terrain elevations above a geoid, an ellipsoid, or other reference surface)

Choosing Units of Length

Using the toolbox effectively depends on being consistent about units of length. Depending on the specific function and the way you are calling it, when you specify lengths, you could be

- Explicitly specifying a radius, reference spheroid object, or ellipsoid vector
- Relying on the function itself to specify a default radius or ellipsoid
- Relying on the reference ellipsoid associated with a map projection structure (`mstruct`)

Whenever you are doing a computation that involves a reference sphere or ellipsoid, make sure that the units of length you are using are the same units used to define the radius of the sphere or semimajor axis of the ellipsoid. These considerations are discussed below.

Converting Units of Length

The following Mapping Toolbox functions convert between different units of length:

- `unitsratio` computes multiplicative factors for converting between 12 different units of length as well as between degrees and radians. You can use `unitsratio` to perform conversions when neither the input units of length nor the output units of length are known until run time. For example,

```
fromUnits = 'radians';
toUnits = 'degrees';
piInDegrees = unitsratio(toUnits, fromUnits) * pi

piInDegrees =

   180
```

- `km2nm`, `km2sm`, `nm2km`, `nm2sm`, `sm2km`, and `sm2nm` perform simple and convenient conversions between kilometers, nautical miles, and statute miles.

These utility functions accept scalars, vectors, and matrices, or any shape. For an overview of these functions and angle conversion functions, see “Summary: Available Distance and Angle Conversion Functions” on page 8-36.

Distances on the Sphere

In this section...
“Arc Length as an Angle in the distance and reckon Functions” on page 8-36
“Summary: Available Distance and Angle Conversion Functions” on page 8-36

Many geospatial domains (seismology, for example) describe distances between points on the surface of the earth as angles. This is simply the result of dividing the length of the shortest great-circle arc connecting a pair points by the radius of the Earth (or whatever planet one is measuring). This gives the angle (in radians) subtended by rays from each point that join at the center of the Earth (or other planet). This is sometimes called a "spherical distance." You can thus call the resulting number a "distance in radians." You could also call the same number a "distance in earth radii." When you work with transformations of geodata, keep this in mind.

You can easily convert that angle from radians to degrees. For example, you can call `distance` to compute the distance in meters from London to Kuala Lumpur:

```
latL =  51.5188;
lonL =  -0.1300;
latK =   2.9519;
lonK = 101.8200;
earthRadiusInMeters = 6371000;
distInMeters = distance(latL, lonL,...
                latK, lonK, earthRadiusInMeters)

distInMeters =
  1.0571e+007
```

Then convert the result to an angle in radians:

```
distInRadians = distInMeters / earthRadiusInMeters

distInRadians =
    1.6593
```

Finally, convert to an angle in degrees:

```
distInDegrees = rad2deg(distInRadians)

distInDegrees =
   95.0692
```

This really only makes sense and produces accurate results when we approximate the Earth (or planet) as a sphere. On an ellipsoid, one can only describe the distance along a geodesic curve using a unit of length.

Mapping Toolbox software includes a set of six functions to conveniently convert distances along the surface of the Earth (or another planet) from units of kilometers (km), nautical miles (nm), or statute miles (sm) to spherical distances in degrees (deg) or radians (rad):

- `km2deg`, `nm2deg`, and `sm2deg` go from length to angle in degrees
- `km2rad`, `nm2rad`, and `sm2rad` go from length to angle in radians

You could replace the final two steps in the preceding example with

```
distInKilometers = distInMeters/1000;
earthRadiusInKm = 6371;
km2deg(distInKilometers, earthRadiusInKm)

ans =
   95.0692
```

Because these conversion can be reversed, the toolbox includes another six convenience functions that convert an angle subtended at the center of a sphere, in degrees or radians, to a great-circle distance along the surface of that sphere:

- `deg2km`, `deg2nm`, and `deg2sm` go from angle in degrees to length
- `rad2km`, `rad2nm`, and `rad2sm` go from angle in radians to length

When given a single input argument, all 12 functions assume a radius of 6,371,000 meters (6371 km, 3440.065 nm, or 3958.748 sm), which is widely-used as an estimate of the average radius of the Earth. An optional second parameter can be used to specify a planetary radius (in output length units) or the name of an object in the Solar System.

Arc Length as an Angle in the distance and reckon Functions

Certain syntaxes of the `distance` and `reckon` functions use angles to denote distances in the way described above. In the following statements, the range argument, `arclen`, is in degrees (along with all the other inputs and outputs):

```
[arclen, az] = distance(lat1, lon1, lat2, lon2)
[latout, lonout] = reckon(lat, lon, arclen, az)
```

By adding the optional `units` argument, you can use radians instead:

```
[arclen, az] = distance(lat1, lon1, lat2, lon2, 'radians')
[latout, lonout] = reckon(lat, lon, arclen, az, 'radians')
```

If an `ellipsoid` argument is provided, however, then `arclen` has units of length, and they match the units of the semimajor axis length of the reference ellipsoid. If you specify `ellipsoid = [1 0]` (the unit sphere), `arclen` can be considered to be either an angle in radians or a length defined in units of earth radii. It has the same value either way. Thus, in the following computation, `lat1`, `lon1`, `lat2`, `lon2`, and `az` are in degrees, but `arclen` will appear to be in radians:

```
[arclen, az] = distance(lat1, lon1, lat2, lon2, [1 0])
```

Summary: Available Distance and Angle Conversion Functions

The following table shows the Mapping Toolbox unit-to-unit distance and arc conversion functions. They all accept scalar, vector, and higher-dimension inputs. The first two columns and rows involve angle units, the last three involve distance units:

Functions that Directly Convert Angles, Lengths, and Spherical Distances

Convert	To Degrees	To Radians	To Kilometers	To Nautical Miles	To Statute Miles

Degrees	`toDegrees`
	`fromDegrees`
		`deg2rad`
		`toRadians`
		`fromDegrees`
			`deg2km`	`deg2nm`	`deg2sm`

Radians	`rad2deg`
	`toDegrees`
	`fromRadians`
		`toRadians`
		`fromRadians`
			`rad2km`	`rad2nm`	`rad2sm`

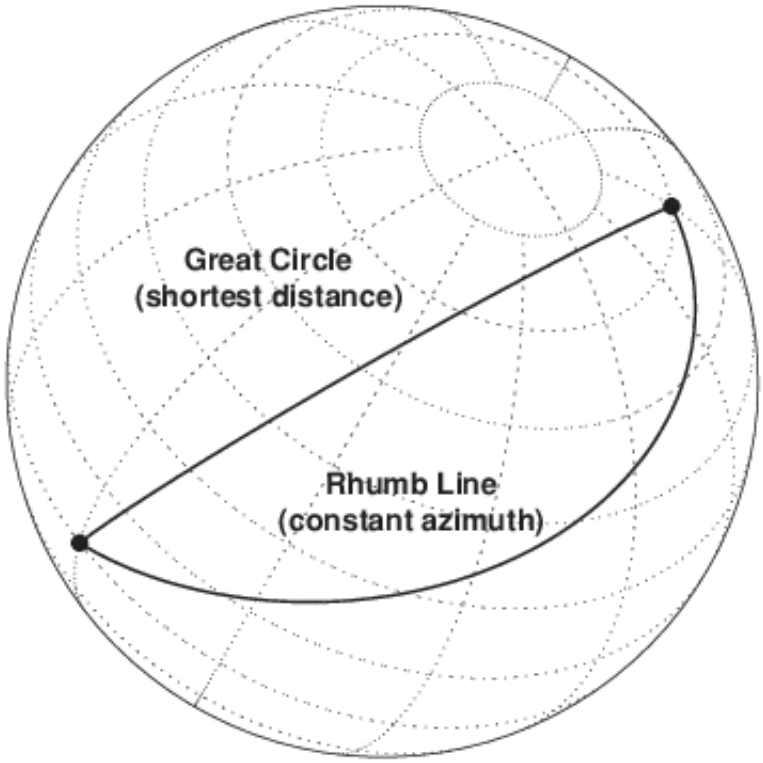
Kilometers	`km2deg`	`km2rad`		`km2nm`	`km2sm`
Nautical Miles	`nm2deg`	`nm2rad`	`nm2km`		`nm2sm`
Statute Miles	`sm2deg`	`sm2rad`	`sm2km`	`sm2nm`	

The angle conversion functions along the major diagonal, `toDegrees`, `toRadians`, `fromDegrees`, and `fromRadians`, can have no-op results. They are intended for use in applications that have no prior knowledge of what angle units might be input or desired as output.

Comparison of Rhumb Lines and Great Circles

A line in a plane has constant slope and represents the shortest path between two points. A line on a sphere can have constant slope or can represent the shortest path between two points, but not necessarily both. If a line on a sphere has constant slope (constant azimuth), then the line follows a rhumb line path. If a line on a sphere represents the shortest path between two points, then the line follows a great circle path.

This figure illustrates two distant locations connected by a great circle path and a rhumb line path.



Rhumb Lines

A rhumb line, also known as a loxodrome, is a curve with a constant azimuth. An azimuth is the angle a line makes with a meridian, measured clockwise from north.

All parallels are rhumb lines because they cross meridians at 90° angles. Additionally, all meridians are rhumb lines.

In general, rhumb lines spiral toward one of the poles. If the azimuth of a rhumb line is true east, west, north, or south, then the rhumb line connects with itself to form a small circle or a pair of antipodal meridians.

Rhumb lines are useful for navigation because the bearing (azimuth) does not change along the route. While rhumb line paths are longer than great circle paths, the constant bearing makes rhumb line paths easier to navigate.

Great Circles

A great circle is the shortest path between two points along the surface of a sphere, as defined by the intersection of the surface of the sphere and a plane passing through the center of the sphere. Great circles always bisect the sphere.

The equator and all meridians are both great circles and rhumb lines. Other great circles are not rhumb lines because they do not have a constant azimuth. Instead, they cross successive meridians at different angles.

Great circles are examples of geodesics. A geodesic is the shortest path between two points on a curved surface, such as the shortest path on an ellipsoid.

Great circles are less useful for navigation because, in general, the bearing changes along the route. While great circle paths are shorter than rhumb line paths, the changes in bearing make great circle paths more difficult to navigate.

Use Rhumb Lines and Great Circles in Functions

Several geometric geodesy functions enable you to specify whether a path follows a rhumb line or a great circle. For example, you can generate track points along either a rhumb line or great circle path by using the `track2` function. You can also find the length of the rhumb line or great circle path that connects two points by using the `distance` function.

See Also

Functions
`track1` | `track` | `azimuth` | `reckon`

Generate Small Circles

A small circle is the intersection of a plane with the surface of a sphere, such as a parallel of latitude on a spherical representation of the Earth. You can generate a small circle from the center point and a point on the perimeter by using the `scircle2` function. You can generate a small circle from the center point and the arc length of the radius by using the `scircle1` function.

For this example, create a small circle that surrounds Norway, Sweden, and Finland by using the `scircle2` function and then create an identical small circle by using the `scircle1` function.

First, specify the coordinates of a center point and a point along the perimeter. Find the coordinates of 100 points that make up the small circle by using the `scircle2` function.

```
centerLat = 62;
centerLon = 20;
perimeterLat = 72;
perimeterLon = 19;
[lat2,lon2] = scircle2(centerLat,centerLon, ...
    perimeterLat,perimeterLon);
```

Calculate the arc length in degrees between the center point and the point along the perimeter. Find the coordinates of the same small circle by using the `scircle1` function.

```
[arclen,~] = distance(centerLat,centerLon, ...
    perimeterLat,perimeterLon);
[lat1,lon1] = scircle1(centerLat,centerLon,arclen);
```

Verify that the small circles are identical, within a tolerance.

```
dLat = abs(lat1-lat2);
dLon = abs(lon1-lon2);
all(dLat < 1e-12)
```

```
ans = logical
   1
```

```
all(dLon < 1e-12)
```

```
ans = logical
   1
```

Display the center point, perimeter point, and small circle on the surface of a geographic globe. Change the view by using the `campos` function.

```
uif = uifigure;
g = geoglobe(uif,Terrain="none");
hold(g,"on")

geoplot3(g,centerLat,centerLon,0,"ro",LineWidth=2)
geoplot3(g,perimeterLat,perimeterLon,0,"ro",LineWidth=2)
geoplot3(g,lat1,lon1,0,"r",LineWidth=2)

campos(g,40,-18,10000000)
```

See Also

`scircle1` | `scircle2`

Interactively Create Small Circle and Track Annotations on Maps

You can interactively generate geographic line annotations, such as navigational tracks and small circles. Great circle tracks are the shortest distance between points that, when closed, partition the Earth into equal halves. A small circle is the locus of points at a constant distance from a reference point. Use `trackg` and `scircleg` to create them by clicking on the map. Double-click the tracks or circles to modify the lines. **Shift+click** the tracks to type specific parameters into a control panel. The control panels also allow you to retrieve or set properties of tracks and circles (for instance, great circle distances and small circle radii).

Set up an orthographic view centered over the Pacific Ocean. Use the `coastlines` MAT-file.

```
axesm('ortho','origin',[30 180])
framem;
gridm
load coastlines
plotm(coastlat,coastlon,'k')
```

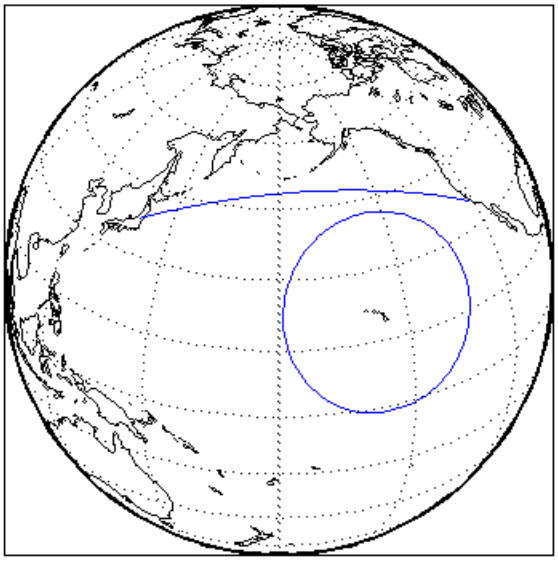
Create a track with the `trackg` function, which prompts for two endpoints. The default track type is a great circle. Create a great circle track from Los Angeles, California, to Tokyo, Japan, and a 1000 km radius small circle centered on the Hawaiian Islands.

```
trackg
Track1:  Click on starting and ending points
```

Now create a small circle around Hawaii with the `scircleg` function, which prompts for a center point and a point on the perimeter. Make the circle's radius about 2000 km, but don't worry about getting the size exact.

```
scircleg
Circle 1:  Click on center and perimeter
```

The map should look approximately like this.



To adjust the size of the small circle to be 2000 km, **Shift+click** anywhere on its perimeter. The Small Circles dialog box appears.

Type `2000` into the **Radius** field.

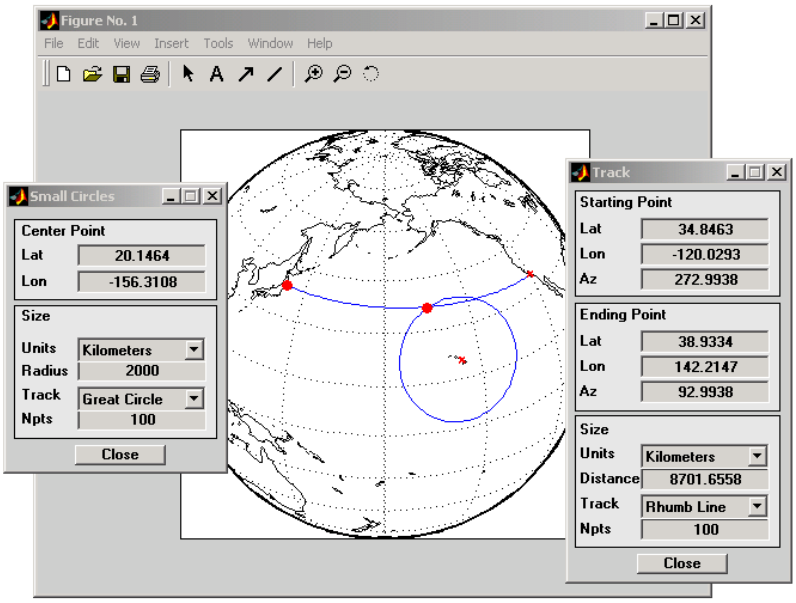
Click **Close**. The small circle adjusts to be 2000 km around Hawaii.

To adjust the track between Los Angeles and Tokyo, **Shift+click** on it. This brings up the Track dialog, with which you specify a position and initial azimuth for either endpoint, as well as the length and type of the track.

Change the track type from `Great Circle` to `Rhumb Line` with the Track pop-up menu. The track immediately changes shape.

Experiment with the other Track dialog controls. Also note that you can move the endpoints of the track with the mouse by dragging the red circles, and obtain the arc's length in various units of distance.

The following figure shows the Small Circles and Track dialog boxes.



Coordinate Reference Systems

- “Summary and Guide to Projections” on page 9-2
- “The Three Main Families of Map Projections” on page 9-15
- “Quantitative Properties of Map Projections” on page 9-19
- “Projection Parameters” on page 9-21
- “Transform Coordinates to a Different Projected CRS” on page 9-27
- “Project and Display Raster Data” on page 9-30
- “Create UTM Maps” on page 9-33
- “Set UTM Parameters Interactively” on page 9-37
- “Work in UTM Without a Displayed Map” on page 9-40
- “Use the Transverse Aspect to Map Across UTM Zones” on page 9-42
- “Map Projections and Distortions” on page 9-44
- “Visualize Spatial Error Using Tissot Indicatrices” on page 9-46
- “Quantify Map Distortions at Point Locations” on page 9-49
- “Projection Aspect” on page 9-53
- “Rotational Transformations on the Globe” on page 9-59

Summary and Guide to Projections

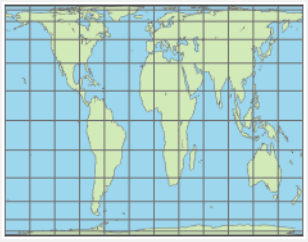
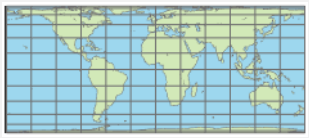
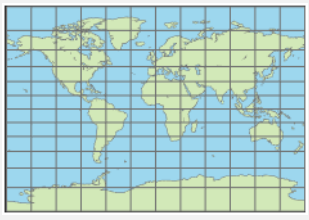
Use projections to display latitude-longitude coordinate data on maps. Choose a projection method by considering these criteria:

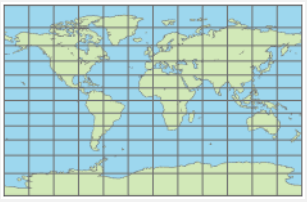
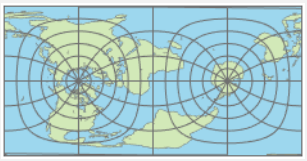
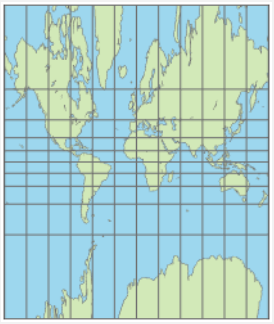
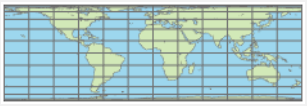
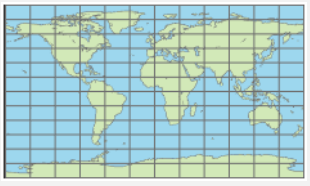
- Family – Choose a cylindrical, conic, or azimuthal projection based on your purpose and region of interest. For more information, see “The Three Main Families of Map Projections” on page 9-15.
- Properties – Choose a projection based on the properties you want to preserve, such as shape, distance, direction, scale, and area. For more information, see “Quantitative Properties of Map Projections” on page 9-19.
- Distortion – Choose a projection based on the distortion you want to minimize or eliminate. For more information, see “Map Projections and Distortions” on page 9-44.

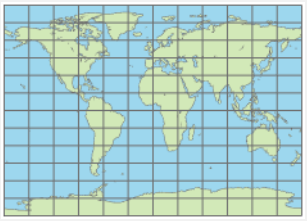
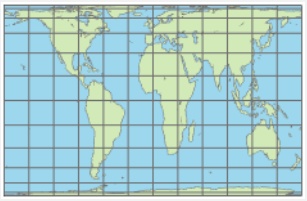
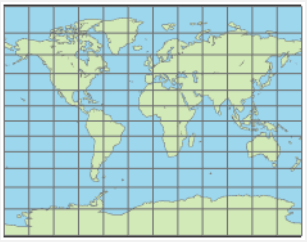
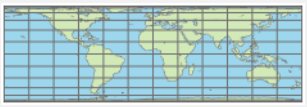
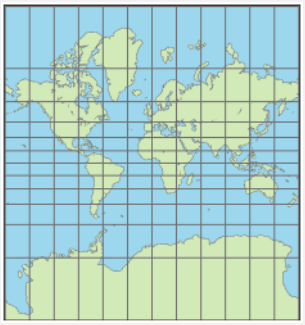
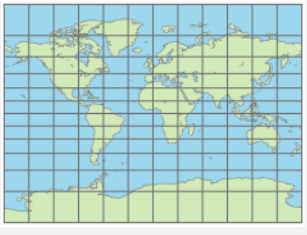
These tables show the map projections you can use with map projection structures and `axesm`-based maps. For more information about map projection structures, see `defaultm`. For more information about `axesm`-based maps, see `axesm`.

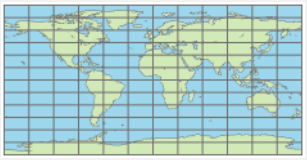
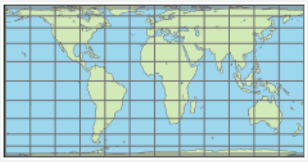
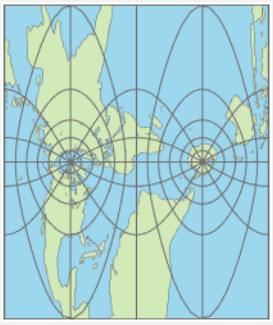
Note Most projection IDs are also functions on the MATLAB search path. These functions are only used in the implementation of functions such as `defaultm` and `axesm`, and therefore their syntaxes are not documented.

Cylindrical Projections

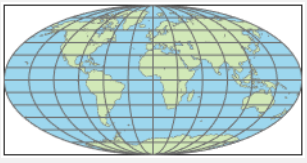
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Balthasart	`balthsrt`	✔	x	x	—	
Behrmann	`behrmann`	✔	x	x	—	
Bolshoi Sovietskii Atlas Mira	`bsam`	x	x	x	—	

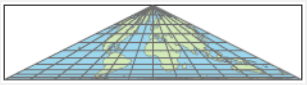
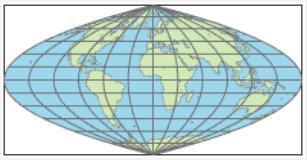
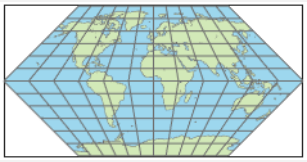
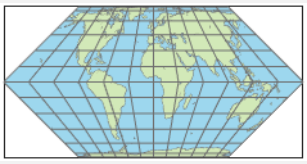
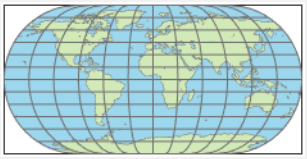
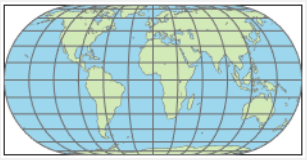
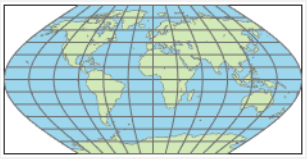
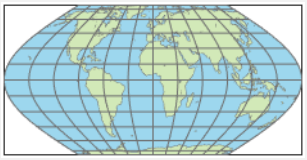
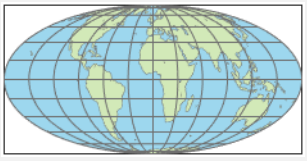
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Braun Perspective	`braun`	x	x	x	—	
Cassini	`cassini`	x	x	✔	—	
Cassini – Standard	`cassinistd`	x	x	x	—	
Central	`ccylin`	x	x	x	—	
Equal-Area Cylindrical	`eqacylin`	✔	x	x	—	
Equidistant Cylindrical	`eqdcylin`	x	x	✔	—	

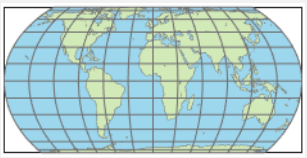
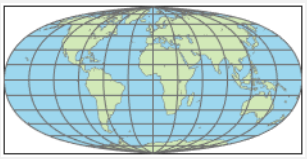
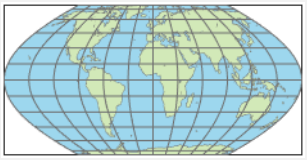
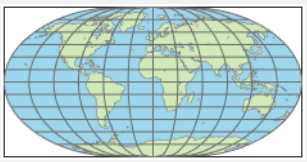
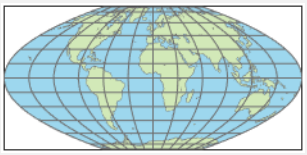
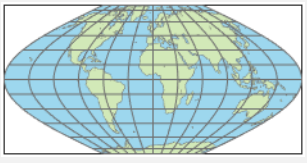
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Gall Isographic	`giso`	x	x	✔	—	
Gall Orthographic	`gortho`	✔	x	x	—	
Gall Stereographic	`gstereo`	x	x	x	—	
Lambert Equal-Area Cylindrical	`lambcyln`	✔	x	x	—	
Mercator	`mercator`	x	✔	x	Rhumb lines are straight.	
Miller	`miller`	x	x	x	—	

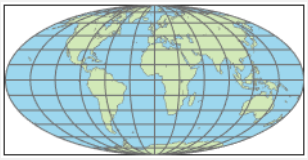
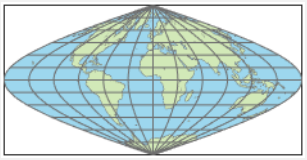
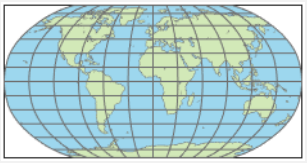
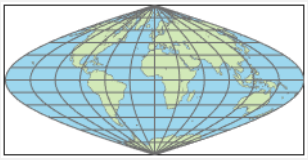
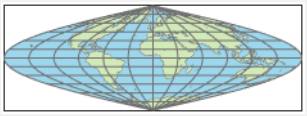
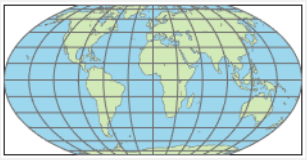
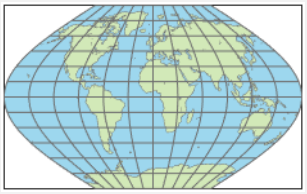
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Plate Carrée	`pcarree`	x	x	✔	—	
Transverse Mercator	`tranmerc`	x	✔	x	—	
Trystan Edwards	`trystan`	✔	x	x	—	
Universal Transverse Mercator (UTM)	`utm`	x	✔	x	—	—
Wetch	`wetch`	x	x	x	—	

Pseudocylindrical Projections

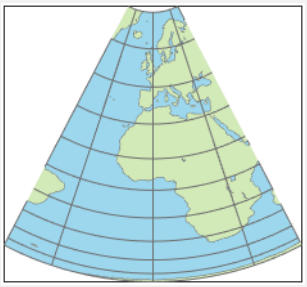
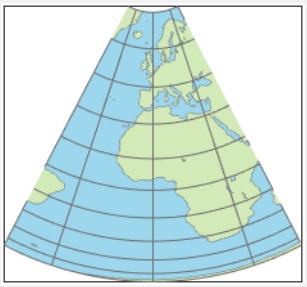
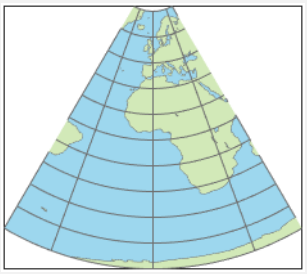
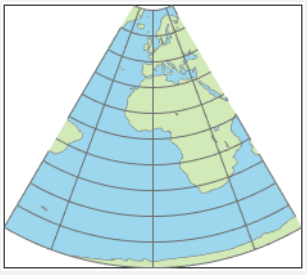
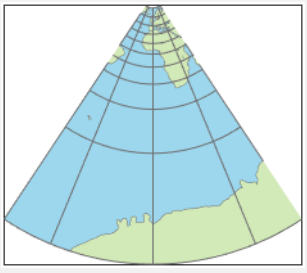
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Apianus II	`apianus`	x	x	x	—	

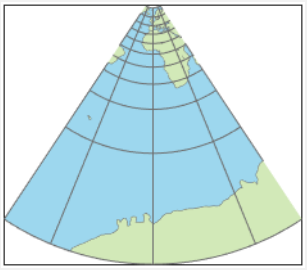
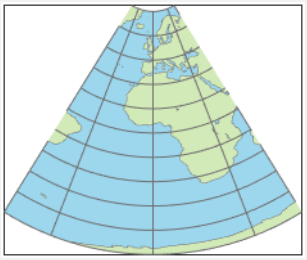
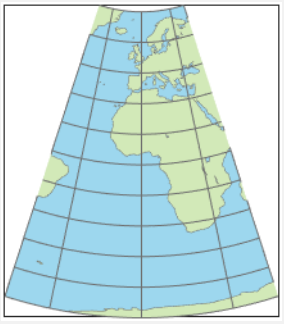
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Collignon	`collig`	✔	x	x	—	
Craster Parabolic	`craster`	✔	x	x	—	
Eckert I	`eckert1`	x	x	x	—	
Eckert II	`eckert2`	✔	x	x	—	
Eckert III	`eckert3`	x	x	x	—	
Eckert IV	`eckert4`	✔	x	x	—	
Eckert V	`eckert5`	x	x	x	—	
Eckert VI	`eckert6`	✔	x	x	—	
Fournier	`fournier`	✔	x	x	—	

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Goode Homolosine	`goode`	✔	x	x	—	
Hatano Asymmetrical Equal-Area	`hatano`	✔	x	x	—	
Kavraisky V	`kavrsky5`	✔	x	x	—	
Kavraisky VI	`kavrsky6`	✔	x	x	—	
Loximuthal	`loximuth`	x	x	x	Rhumb lines from the central point are straight, true to scale, and correct in azimuth.	
McBryde-Thomas Flat-Polar Parabolic	`flatplrp`	✔	x	x	—	
McBryde-Thomas Flat-Polar Quartic	`flatplrq`	✔	x	x	—	
McBryde-Thomas Flat-Polar Sinusoidal	`flatplrs`	✔	x	x	—	

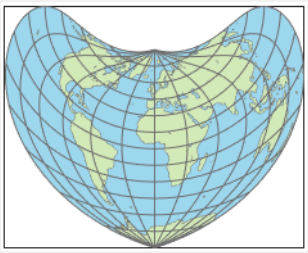
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Mollweide	`mollweid`	✔	x	x	—	
Putnins P5	`putnins5`	x	x	x	—	
Quartic Authalic	`quartic`	✔	x	x	—	
Robinson	`robinson`	x	x	x	—	
Sinusoidal	`sinusoid`	✔	x	x	—	
Tissot Modified Sinusoidal	`modsine`	✔	x	x	—	
Wagner IV	`wagner4`	✔	x	x	—	
Winkel 1	`winkel`	x	x	x	—	

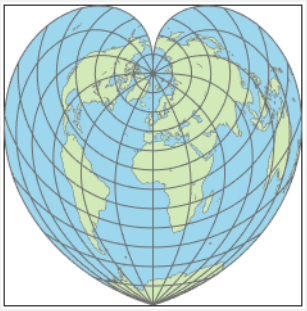
Conic Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Albers Equal-Area Conic	`eqaconic`	✔	x	x	—	
Albers Equal-Area Conic – Standard	`eqaconicstd`	✔	x	x	—	
Equidistant Conic	`eqdconic`	x	x	✔	—	
Equidistant Conic – Standard	`eqdconicstd`	x	x	✔	—	
Lambert Conformal Conic	`lambert`	x	✔	x	—	

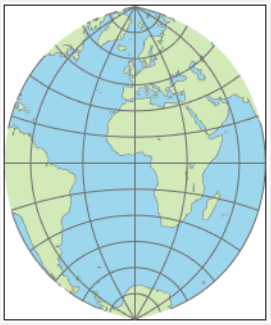
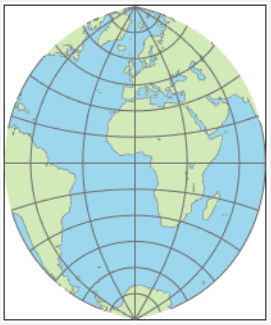
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Lambert Conformal Conic – Standard	`lambertstd`	x	✔	x	—	
Murdoch I Conic	`murdoch1`	x	x	✔	The total area is correct.	
Murdoch III Minimum Error Conic	`murdoch3`	x	x	✔	The total area is correct.	

Pseudoconic Projections

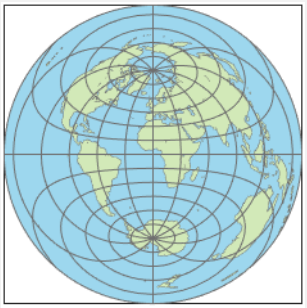
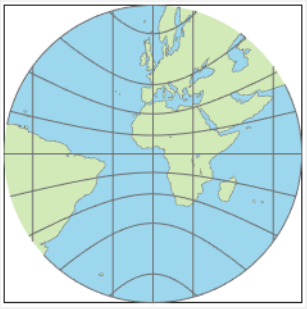
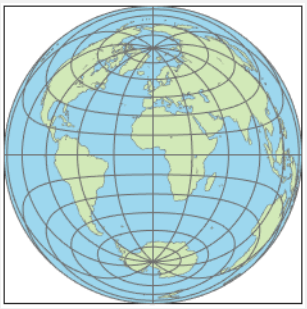
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Bonne	`bonne`	✔	x	x	—	

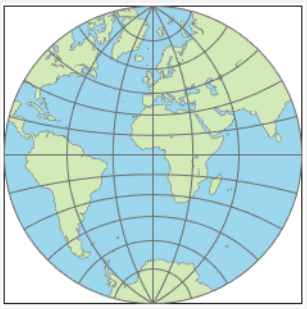
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Werner	`werner`	✔	x	x	—	

Polyconic Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Polyconic	`polycon`	x	x	x	—	
Polyconic – Standard	`polyconstd`	x	x	x	—	
Van Der Grinten I	`vgrint1`	x	x	x	—	

Azimuthal Projections

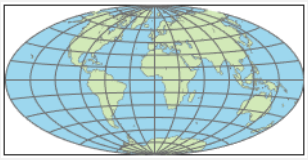
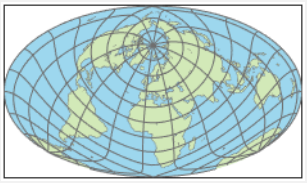
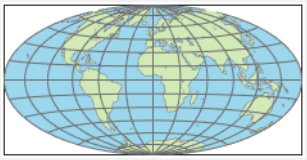
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Breusing Harmonic Mean	`breusing`	x	x	x	—	
Equidistant Azimuthal	`eqdazim`	x	x	✔	—	
Gnomonic	`gnomonic`	x	x	x	Great circles appear as straight lines.	
Lambert Azimuthal Equal-Area	`eqaazim`	✔	x	x	—	

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Orthographic	`ortho`	x	x	x	—	
Stereographic	`stereo`	x	✔	x	Great and small circles appear as either straight lines or circular arcs.	
Universal Polar Stereographic (UPS)	`ups`	x	✔	x	Great and small circles appear as either straight lines or circular arcs.	—
Vertical Perspective Azimuthal	`vperspec`	x	x	x	—	

Pseudoazimuthal Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Wiechel	`wiechel`	✔	x	x	—	

Modified Azimuthal Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Aitoff	`aitoff`	x	x	x	—	
Briesemeister	`bries`	✔	x	x	—	
Hammer	`hammer`	✔	x	x	—	

See Also

`axesm` | `geoshow` | `defaultm`

The Three Main Families of Map Projections

In this section...
“Unwrapping the Sphere to a Plane” on page 9-15
“Cylindrical Projections” on page 9-15
“Conic Projections” on page 9-16
“Azimuthal Projections” on page 9-17

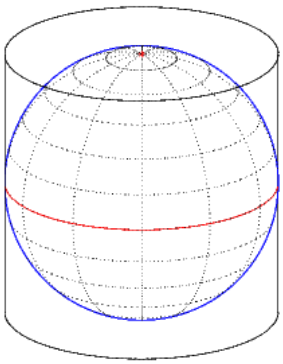
Unwrapping the Sphere to a Plane

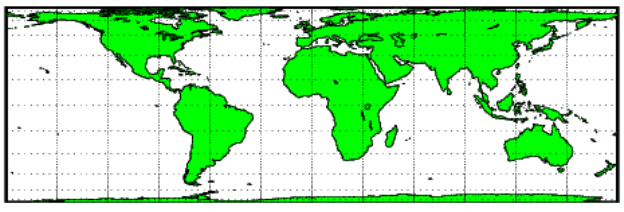
Mapmakers have developed hundreds of map projections, over several thousand years. Three large families of map projection, plus several smaller ones, are generally acknowledged. These are based on the types of geometric shapes that are used to transfer features from a sphere or spheroid to a plane. Map projections are based on *developable surfaces*, and the three traditional families consist of cylinders, cones, and planes. They are used to classify the majority of projections, including some that are not analytically (geometrically) constructed. In addition, a number of map projections are based on polyhedra. While polyhedral projections have interesting and useful properties, they are not described in this guide.

Which developable surface to use for a projection depends on what region is to be mapped, its geographical extent, and the geometric properties that areas, boundaries, and routes need to have, given the purpose of the map. The following sections describe and illustrate how the cylindrical, conic, and azimuthal families of map projections are constructed and provides some examples of projections that are based on them.

Cylindrical Projections

A *cylindrical* projection is produced by wrapping a cylinder around a globe representing the Earth. The map projection is the image of the globe projected onto the cylindrical surface, which is then unwrapped into a flat surface. When the cylinder aligns with the polar axis, parallels appear as horizontal lines and meridians as vertical lines. Cylindrical projections can be either equal-area, conformal, or equidistant. The following figure shows a regular cylindrical or *normal aspect* orientation in which the cylinder is tangent to the Earth along the Equator and the projection radiates horizontally from the axis of rotation. The projection method is diagrammed on the left, and an example is given on the right (equal-area cylindrical projection, normal/equatorial aspect).





For a description of projection aspect, see “Projection Aspect” on page 9-53.

Some widely used cylindrical map projections are

- Equal-area cylindrical projection
- Equidistant cylindrical projection
- Mercator projection
- Miller projection
- Plate Carrée projection
- Universal transverse Mercator projection

Pseudocylindrical Map Projections

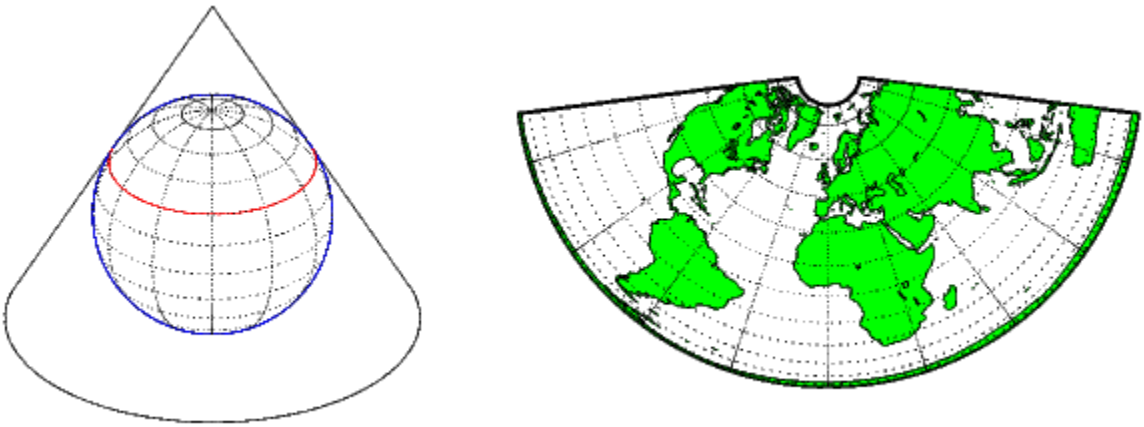
All cylindrical projections fill a rectangular plane. *Pseudocylindrical* projection outlines tend to be barrel-shaped rather than rectangular. However, they do resemble cylindrical projections, with straight and parallel latitude lines, and can have equally spaced meridians, but meridians are curves, not straight lines. Pseudocylindrical projections can be equal-area, but are not conformal or equidistant.

Some widely-used pseudocylindrical map projections are

- Eckert projections (I-VI)
- Goode homolosine projection
- Mollweide projection
- Quartic authalic projection
- Robinson projection
- Sinusoidal projection

Conic Projections

A *conic* projection is derived from the projection of the globe onto a cone placed over it. For the *normal aspect*, the apex of the cone lies on the polar axis of the Earth. If the cone touches the Earth at just one particular parallel of latitude, it is called *tangent*. If made smaller, the cone will intersect the Earth twice, in which case it is called *secant*. Conic projections often achieve less distortion at mid- and high latitudes than cylindrical projections. A further elaboration is the *polyconic* projection, which deploys a family of tangent or secant cones to bracket a succession of bands of parallels to yield even less scale distortion. The following figure illustrates conic projection, diagramming its construction on the left, with an example on the right (Albers equal-area projection, polar aspect).

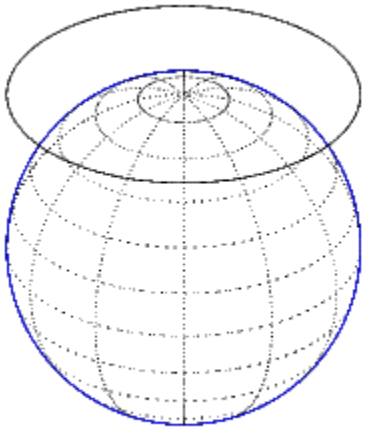


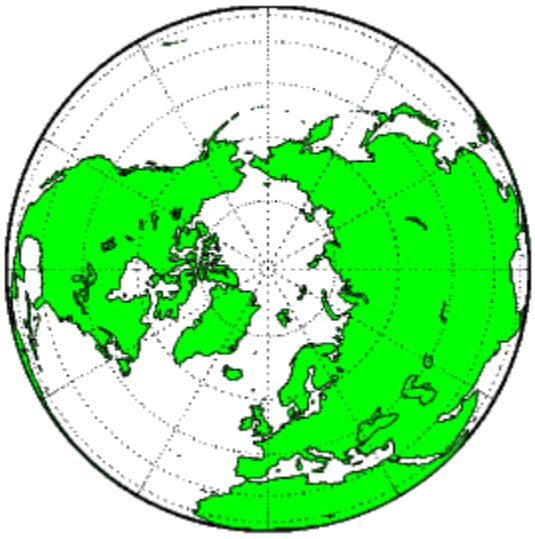
Some widely-used conic projections are

- Albers Equal-area projection
- Equidistant projection
- Lambert conformal projection
- Polyconic projection

Azimuthal Projections

An *azimuthal* projection is a projection of the globe onto a plane. In polar aspect, an azimuthal projection maps to a plane tangent to the Earth at one of the poles, with meridians projected as straight lines radiating from the pole, and parallels shown as complete circles centered at the pole. Azimuthal projections (especially the orthographic) can have equatorial or oblique aspects. The projection is centered on a point, that is either on the surface, at the center of the Earth, at the antipode, some distance beyond the Earth, or at infinity. Most azimuthal projections are not suitable for displaying the entire Earth in one view, but give a sense of the globe. The following figure illustrates azimuthal projection, diagramming it on the left, with an example on the right (orthographic projection, polar aspect).





Some widely used azimuthal projections are

- Equidistant azimuthal projection
- Gnomonic projection
- Lambert equal-area azimuthal projection
- Orthographic projection
- Stereographic projection
- Universal polar stereographic projection

Quantitative Properties of Map Projections

A sphere, unlike a polyhedron, cone, or cylinder, cannot be reformed into a plane. To portray the surface of a sphere on a plane, you must first define a developable surface (a surface that you can cut and flatten onto a plane without stretching or creasing it), and then create rules for representing part or all of the surface on the plane.

All projections result in distortion. Depending on the projection, you can preserve these map properties: shape, distance, direction, scale, and area. No projection can retain more than one of these properties over a large portion of a sphere.

For a description of the properties that specific map projections preserve, see “Summary and Guide to Projections” on page 9-2.

Shape

Map projections that preserve shape locally (within "small" areas) are conformal. A projection is conformal when, at any point, the scale of the map is the same in all directions. When a projection preserves shape, the meridians (lines of longitude) and parallels (lines of latitude) intersect at right angles. An older term for conformal is orthomorphic.

Distance

Map projections that preserve distance are equidistant. A projection is equidistant when it preserves distances from the center of the projection to all other points on the map. Maps are also described as equidistant when the separation between the parallels is uniform (as in, the projection maintains distances along meridians). No map projection maintains distance in all directions from any arbitrary point.

Direction

A map projection preserves direction when the projection portrays azimuths (angles from the central point or from a point on a line to another point) correctly in all directions. Many azimuthal projections preserve direction.

Scale

Scale is the ratio between a distance portrayed on a map and the same extent on the sphere. No projection maintains constant scale over large areas, but some are able to limit scale variation to one or two percent.

Area

Map projections that preserve area are equal-area or equivalent. Equal-area projections portray areas in proportional relationships to the areas on the sphere that they represent. Older terms for equal-area are homolographic, homalographic, authalic, and equireal. No map can be both equal-area and conformal.

See Also

Functions
`projfwd` | `projinv`

Objects
`projcrs`

Related Examples

- “The Three Main Families of Map Projections” on page 9-15

Projection Parameters

Every projection has at least one parameter that controls how it transforms geographic coordinates into planar coordinates. Some projections are rather fixed, and aside from the orientation vector and nominal scale factor, have no parameters that the user should vary, as to do so would violate the definition of the projection. For example, the Robinson projection has one standard parallel that is fixed by definition at 38° North and South; the Cassini and Wetch projections cannot be constructed in other than Normal aspect. In general, however, projections have several variable parameters. The following section discusses map projection parameters and provides guidance for setting them.

Projection Characteristics Maps Can Have

In addition to the name of the projection itself, the parameters that a map projection can have are

- *Aspect* — Orientation of the projection on the display surface
- *Center* or *Origin* — Latitude and longitude of the midpoint of the display
- *Scale Factor* — Ratio of distance on the map to distance on the ground
- *Standard Parallel(s)* — Chosen latitude(s) where scale distortion is zero
- *False Northing* — Planar offset for coordinates on the vertical map axis
- *False Easting* — Planar offset for coordinates on the horizontal map axis
- *Zone* — Designated latitude-longitude quadrangle used to systematically partition the planet for certain classes of projections

While not all projections require all these parameters, there will always be a projection aspect, origin, and scale.

Other parameters are associated with the graphic expression of a projection, but do not define its mathematical outcome. These include

- Map latitude and longitude limits
- Frame latitude and longitude limits

However, as certain projections are unable to map an entire planet, or become very distorted over large regions, these limits are sometimes a necessary part of setting up a projection.

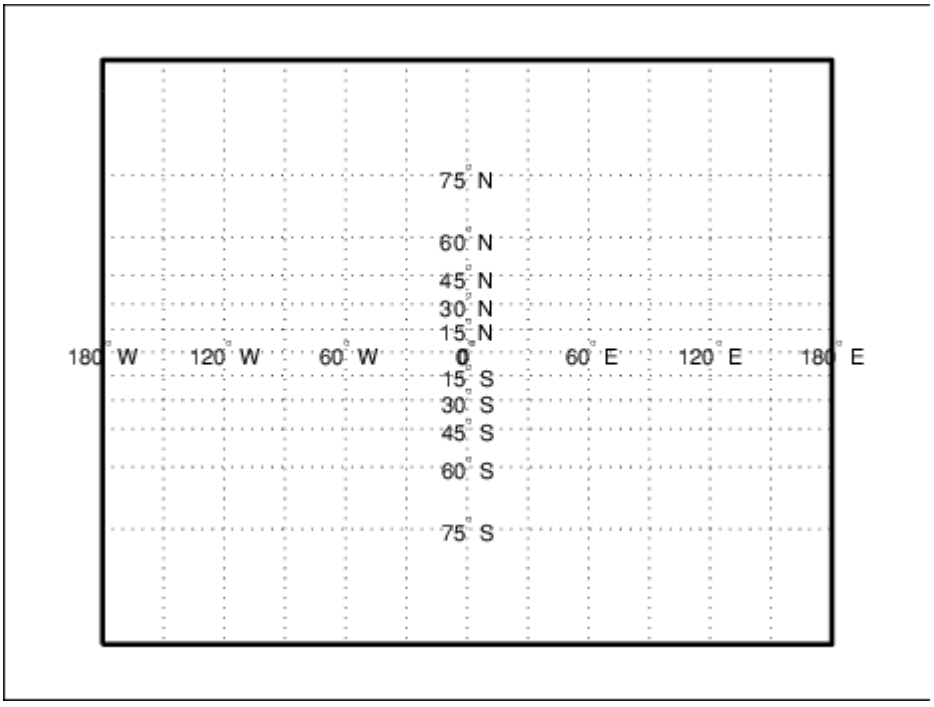
Determining Projection Parameters

In the following exercise, you define an `axesm`-based map and examine default parameters for a cylindrical, a conic, and an azimuthal projection.

1 Set up a default Mercator projection (which is cylindrical) and pass its handle to the `getm` function to query projection parameters:

```
figure;
h=axesm('Mapprojection','mercator','Grid','on','Frame','on',...
'MlabelParallel',0,'PlabelMeridian',0,'mlabellocation',60,...
'meridianlabel','on','parallellabel','on')
```

The graticule and frame for the default map projection are shown below.



2 Query the handle of the `axesm`-based map using `getm` to inspect the properties that pertain to map projection parameters. The principal ones are `aspect`, `origin`, `scalefactor`, `nparallels`, `mapparallels`, `falsenorthing`, `falseeasting`, `zone`, `maplatlimit`, `maplonlimit`, `rlatlimit`, and `flonlimit`:

```
getm(h,'aspect')

ans =
     normal

getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     1

getm(h,'mapparallels')

ans =
     0

getm(h,'falsenorthing')

ans =
```

```
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -86    86

getm(h,'maplonlimit')

ans =
  -180   180

getm(h,'Flatlimit')

ans =
   -86    86

getm(h,'Flonlimit')

ans =
  -180   180
```

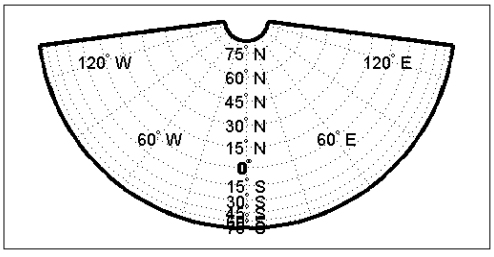
For more information on these and other properties of `axesm`-based maps, see axesm-Based Map Properties.

3 Reset the projection type to equal-area conic (`'eqaconic'`). The figure is redrawn to reflect the change. Determine the parameters that the toolbox changes in response:

```
setm(h,'Mapprojection', 'eqaconic')
getm(h,'aspect')

ans =
normal

getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     2
```

```
getm(h,'mapparallels')

ans =
    15    75

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -86    86

getm(h,'maplonlimit')

ans =
  -135   135

getm(h,'Flatlimit')

ans =
   -86    86

getm(h,'Flonlimit')

ans =
  -135   135
```

The `eqaconic` projection has two standard parallels, at 15° and 75°. It also has reduced longitude limits (covering 270° rather than 360°). The resulting `eqaconic` graticule is shown below.

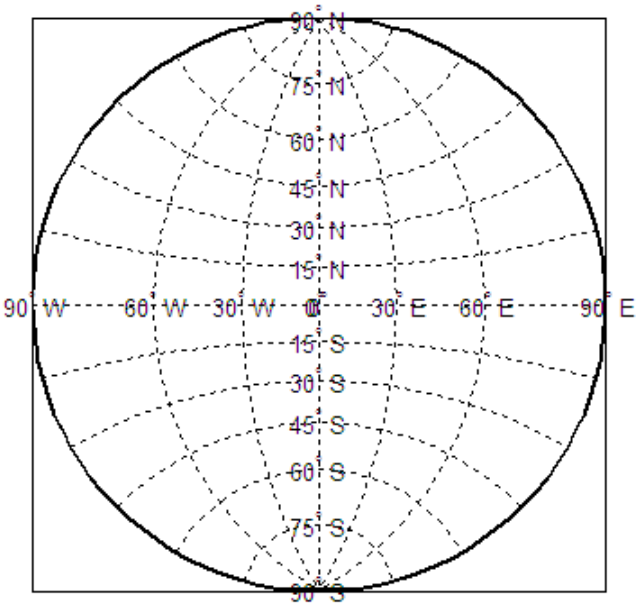


4 Now set the projection type to Stereographic (`'stereo'`) and examine the same properties as you did for the previous projections:

```
setm(h,'Mapprojection','stereo')
setm(gca,'MLabelParallel',0,'PLabelMeridian',0)
getm(h,'aspect')

ans =
normal

getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     0

getm(h,'mapparallels')

ans =
     []

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -90    90

getm(h,'maplonlimit')

ans =
  -180   180

getm(h,'Flatlimit')

ans =
   -Inf    90

getm(h,'Flonlimit')
```

```
ans =
  -180   180
```

The stereographic projection, being azimuthal, does not have standard parallels, so none are indicated. The map limits do not change from the previous projection. The map figure is shown below.



Transform Coordinates to a Different Projected CRS

If you directly compare data sets with different projected coordinate reference systems (CRSs), then the results are inaccurate. Therefore, before comparing data sets, first verify that the CRSs are the same. If different projected CRSs have the same underlying geographic CRS, then you can transform the coordinates from one projected CRS to the other. Once the data sets are referenced to the same projected CRS, you can compare them.

To transform projected *x*-*y* coordinates to a different projected CRS, first unproject the *x*-*y* coordinates to latitude-longitude coordinates by using the `projinv` function. Then, project the latitude-longitude coordinates to *x*-*y* coordinates in a different projected CRS by using the `projfwd` function.

For example, import a shapefile containing the *x*- and *y*-coordinates of roads in Boston. Also import information about the shapefile as a structure. Find the projected CRS for the coordinates by accessing the `CoordinateReferenceSystem` field of that structure.

```
s = shaperead('boston_roads.shp');
x1 = [s.X];
y1 = [s.Y];
info = shapeinfo('boston_roads.shp');
p1 = info.CoordinateReferenceSystem;
```

Unproject the *x*-*y* coordinates and return latitude-longitude coordinates.

```
[lat,lon] = projinv(p1,x1,y1);
```

Select a new projected CRS for the target projection. For this example, create a `projcrs` object for UTM zone 19N. Verify that both projected CRSs have the same geographic CRS. If the geographic CRSs are different, then the projected coordinates may be inaccurate. You can find the geographic CRS by querying the `GeographicCRS` property of the `projcrs` object.

```
p2 = projcrs(26919);
p2.GeographicCRS.Name
```

```
ans =
"NAD83"
```

```
p1.GeographicCRS.Name
```

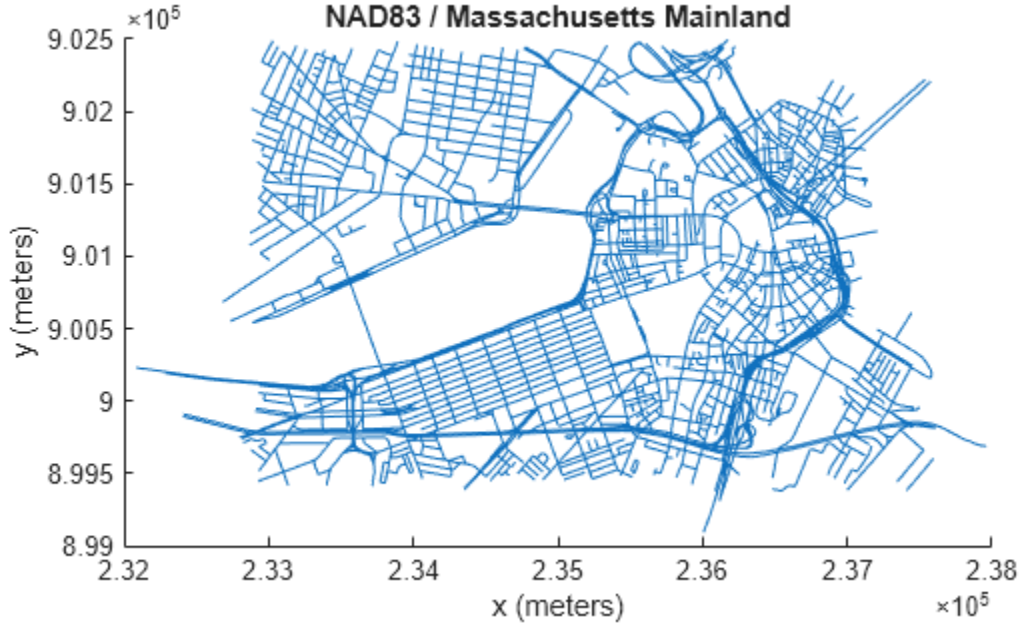
```
ans =
"NAD83"
```

Project the latitude-longitude coordinates to *x*-*y* coordinates by specifying the `projcrs` object you created.

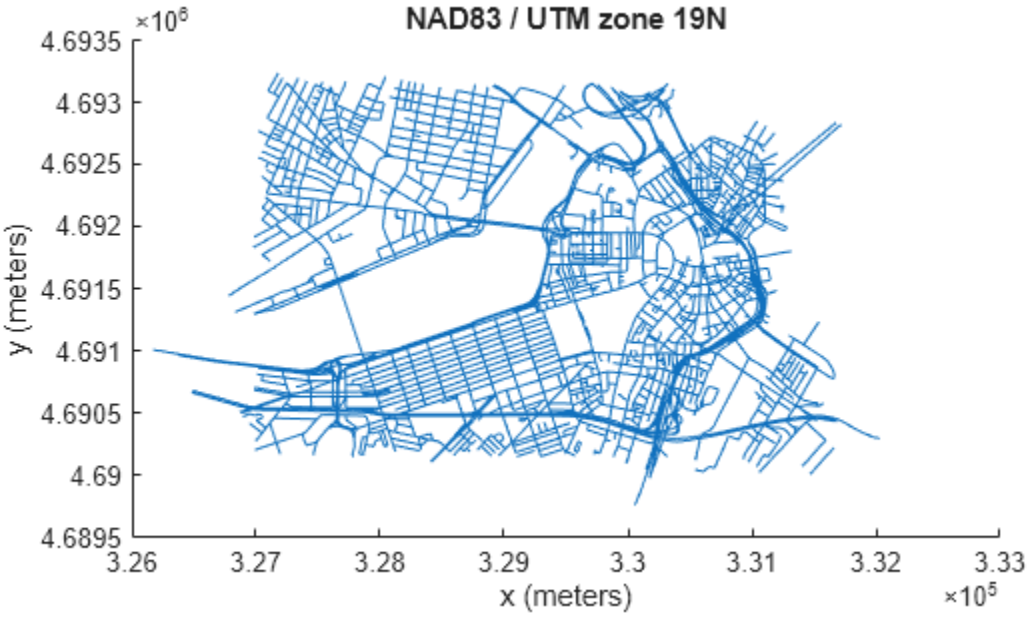
```
[x2,y2] = projfwd(p2,lat,lon);
```

Compare the original *x*-*y* coordinates with the new *x*-*y* coordinates by displaying them. Add labels and a title to each figure.

```
figure
mapshow(x1,y1)
xlabel('x (meters)')
ylabel('y (meters)')
title(p1.Name)
```




```
figure
mapshow(x2,y2)
xlabel('x (meters)')
ylabel('y (meters)')
title(p2.Name)
```



The visualizations are similar, but the coordinates displayed along the axis rulers correspond to different projected CRSs.

See Also

`projinv` | `projfwd` | `geocrs` | `projcrs` | `shaperead`

More About

- “Project and Display Raster Data” on page 9-30

Project and Display Raster Data

To project or unproject regularly-spaced raster data that is associated with a geographic or map reference object, you must first create a coordinate grid that matches the size of the raster. Use different grid creation functions depending on which way you are projecting. When you project latitude-longitude coordinates to *x*-*y* coordinates, create a grid using the `geographicGrid` function. When you unproject *x*-*y* coordinates to latitude-longitude coordinates, create a grid using the `worldGrid` function.

After transforming the raster data, you can display it on a map using visualization functions such as `mapshow` and `geoshow`. Use `mapshow` for projected *x*-*y* coordinates and `geoshow` for unprojected latitude-longitude coordinates.

Project Raster Data

To project data that is associated with a geographic raster reference object, first create a grid of latitude-longitude coordinates for each point in the raster. Then, project the geographic coordinates to *x*-*y* map coordinates.

For example, import elevation raster data as an array and a geographic cells reference object. Get the latitude-longitude coordinates for each point in the raster by using the `geographicGrid` function.

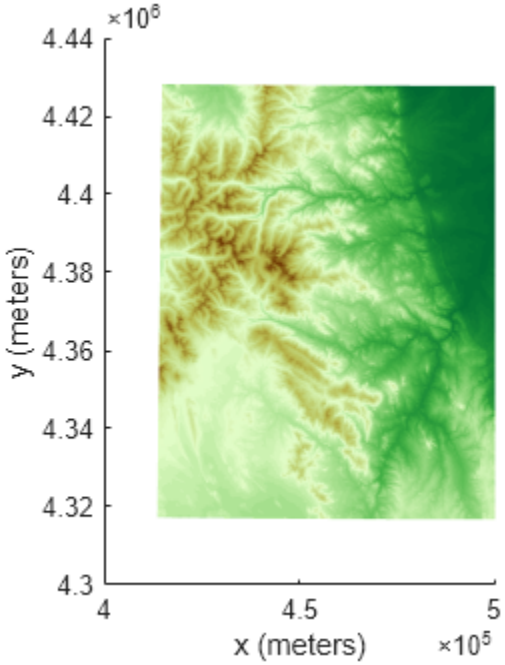
```
[Z,R] = readgeoraster('n39_w106_3arc_v2.dt1');
[lat,lon] = geographicGrid(R);
```

Now that you have your grid, select a map projection to use when projecting the coordinates. For this example, create a `projcrs` object for UTM zone 13 in the northern hemisphere. Then, project the latitude-longitude coordinates to *x*-*y* coordinates.

```
p = projcrs(32613);
[x,y] = projfwd(p,lat,lon);
```

Display the projected raster as a surface by calling `mapshow` and specifying the *x*-*y* coordinates and elevation array. Add axis labels and apply a colormap appropriate for elevation data.

```
figure
mapshow(x,y,Z,'DisplayType','surface')
xlabel('x (meters)')
ylabel('y (meters)')
demcmap(Z)
```



If the geographic CRS of the latitude-longitude coordinates does not match the geographic CRS of the projected CRS, then the projected coordinates may be inaccurate. You can find the geographic CRS of a `projcrs` object or a geographic raster reference object by querying their `GeographicCRS` properties.

```
p.GeographicCRS.Name
```

```
ans =
"WGS 84"
```

```
R.GeographicCRS.Name
```

```
ans =
"WGS 84"
```

The DTED file used in this example is courtesy of the US Geological Survey.

Unproject Raster Data

To unproject data that is associated with a map raster reference object, first create a grid of *x*-*y* coordinates for each point in the raster. Then, unproject the *x*-*y* map coordinates to geographic coordinates.

For example, import an image of Boston as an array and a map cells reference object. Get information about the map projection as a `projcrs` object by querying the `ProjectedCRS` property of the reference object.

```
[Z,R] = readgeoraster('boston.tif');
p = R.ProjectedCRS;
```

Get the *x*-*y* coordinates for each point in the raster by using the `worldGrid` function.

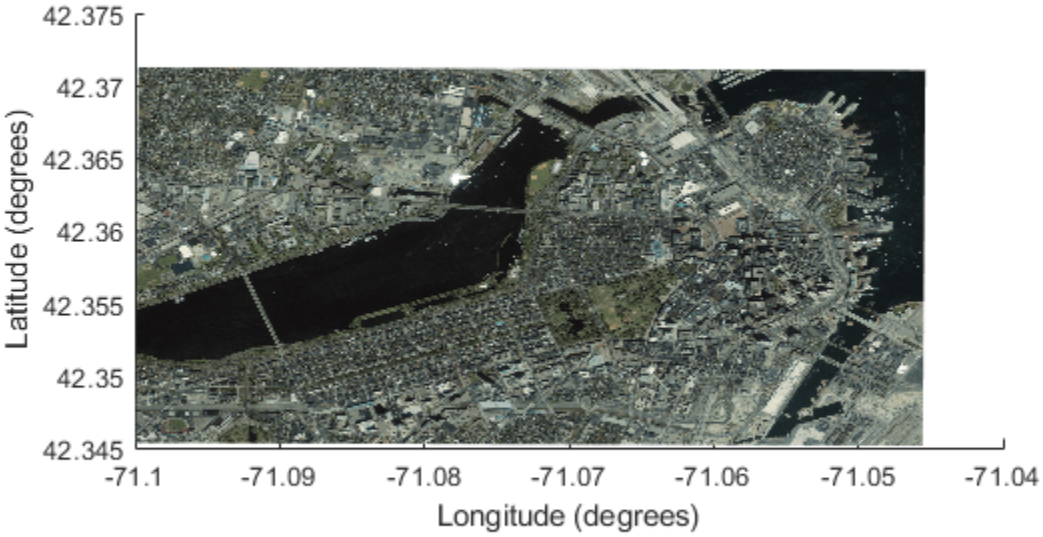
```
[x,y] = worldGrid(R);
```

Unproject the *x*-*y* coordinates to latitude-longitude coordinates by using the `projinv` function and specifying the `projcrs` object and coordinate grid.

```
[lat,lon] = projinv(p,x,y);
```

Display the unprojected image by calling `geoshow` and specifying the latitude-longitude coordinates and image array. By default, `geoshow` displays coordinates using a Plate Carrée projection. Then, add axis labels.

```
figure
geoshow(lat,lon,Z)
xlabel('Longitude (degrees)')
ylabel('Latitude (degrees)')
```



See Also

Functions
`projinv` | `projfwd` | `intrinsicToWorld` | `meshgrid` | `worldGrid`

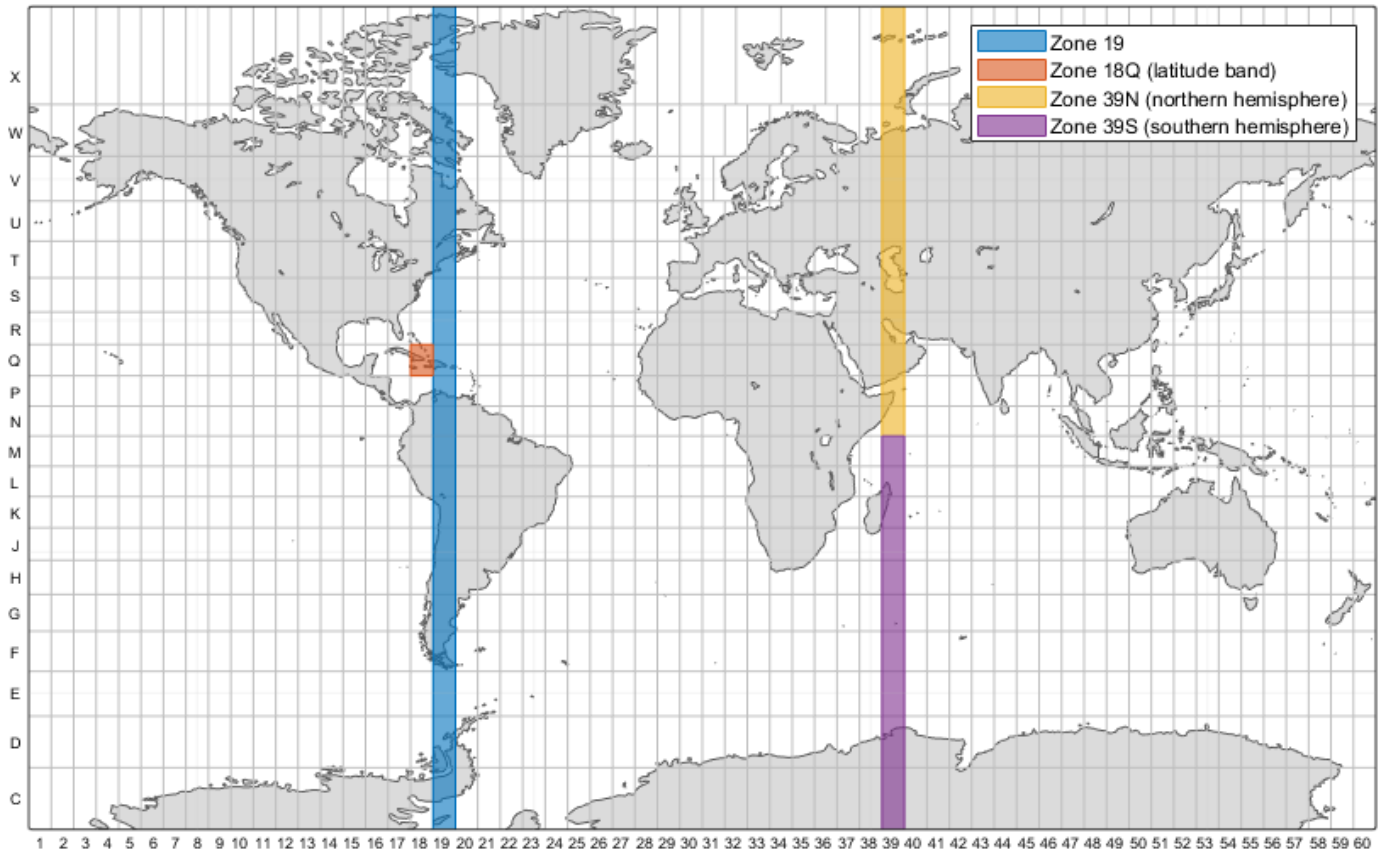
Objects
`MapCellsReference` | `GeographicCellsReference`

Create UTM Maps

The Universal Transverse Mercator (UTM) system divides the world into a set of quadrangles called zones. Each zone uses a transverse Mercator projection that is designed to limit distortion. Most zones cover 6 degrees of longitude. The UTM system is defined for latitudes between –80 and 84 degrees. Beyond these latitudes, use the Universal Polar Stereographic (UPS) system, instead.

Mapping Toolbox functions identify zones using zone numbers from 1 to 60. The functions use different conventions to identify regions within the zones.

- For `projcrs` objects created from EPSG codes, zone numbers with hemisphere letters indicate either the northern or southern hemisphere. The image shows zone 39N in yellow and zone 39S in purple.
- For `axesm`-based maps, the `utmzone` function, and the `utmzoneui` function, zone numbers indicate zones across both the northern and southern hemispheres. Zone numbers with latitude band letters indicate regions that, in most cases, cover 8 degrees of latitude. Some zones in northern Europe cover more or less than 6 degrees of longitude. The image shows zone 19 in blue and zone 18Q in orange.



This topic shows how to create similar maps using map axes and `axesm`-based maps. For a comparison of map axes and `axesm`-based maps, including when to use each type of display, see “Choose a 2-D Map Display” on page 3-2.

Create UTM Map Using Map Axes

Read a shapefile of US states into the workspace as a geospatial table. The shapefile represents the states using polygon shapes in geographic coordinates.

```
states = readgeotable("usastatelo.shp");
```

Create a subtable that includes New York and the states that comprise New England: Maine, New Hampshire, Vermont, Massachusetts, Connecticut, and Rhode Island.

```
names = ["New York","Maine","New Hampshire","Vermont","Massachusetts", ...
    "Connecticut","Rhode Island"];
NEstates = geocode(names,states);
```

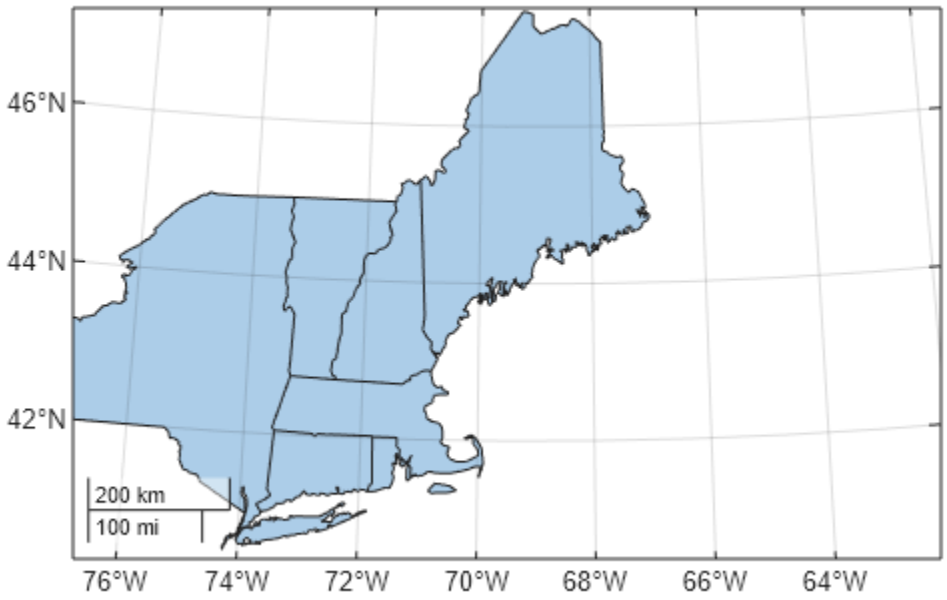
Display Data in Zone and Surrounding Area

Create a projected coordinate reference system (CRS) object that is appropriate for New England. Use the NAD83 / UTM zone 19N projected CRS, which has the EPSG code 26919. For this CRS, zone 19N indicates zone 19 in the northern hemisphere.

```
proj = projcrs(26919);
```

Create a new map that uses the projected CRS. Then, plot the states.

```
figure
newmap(proj)
geoplot(NEstates)
```



Display Only Data in Zone

By default, the map displays data in the box specified by the `Position` property of the axes object. As a result, the map can display data within the UTM zone and in the surrounding longitudes. To display data within only the UTM zone, use a cartographic map layout.

Create a new map that uses the same projected CRS.

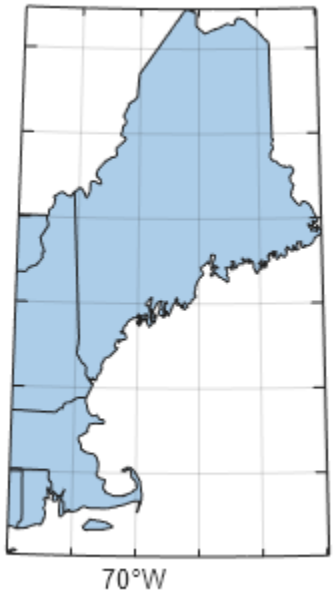
```
figure
mx = newmap(proj);
```

Get the cartographic latitude and longitude limits of the CRS. Then, clip the states to the limits.

```
latlim = mx.CartographicLatitudeLimits;
lonlim = mx.CartographicLongitudeLimits;
clipped = geoclip(NEstates.Shape,latlim,lonlim);
```

Display the clipped shapes on the map. Then, change the map layout to cartographic.

```
geoplot(clipped)
mx.MapLayout = "cartographic";
```



Create UTM Map Using axesm-Based Maps

Read a shapefile of US states into the workspace as a geospatial table. The shapefile represents the states using polygon shapes in geographic coordinates.

```
states = readgeotable("usastatelo.shp");
```

Create a subtable that includes New York and the states that comprise New England: Maine, New Hampshire, Vermont, Massachusetts, Connecticut, and Rhode Island.

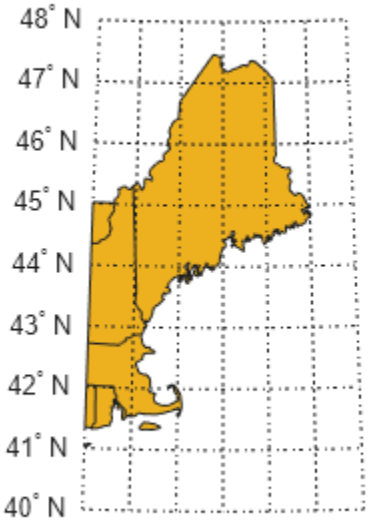
```
names = ["New York","Maine","New Hampshire","Vermont","Massachusetts" ...
    "Connecticut","Rhode Island"];
NEstates = geocode(names,states);
```

Create an `axesm`-based map for UTM zone 19T. Specify the reference ellipsoid as the Geodetic Reference System 1980 (GRS 80). Turn on the grid and parallel labels, and remove the axes lines.

```
figure
axesm("utm",Zone="19T",Geoid=referenceEllipsoid("grs80"))
gridm
plabel
axis off
```

Display the shapes on the map. By default, the map uses the limits for the UTM zone.

```
geoshow(NEstates)
```



See Also

Functions
`utmzone` | `utmzoneui` | `utm` | `ups`

Objects
`projcrs`

Related Examples

- “Create Map of Quadrangle Using Cartographic Map Layout” on page 4-16

Set UTM Parameters Interactively

The easiest way to use the UTM projection is through a graphical user interface. You can create or modify a UTM area of interest with the `axesmui` projection control panel, and get further assistance form the `utmzoneui` control panel.

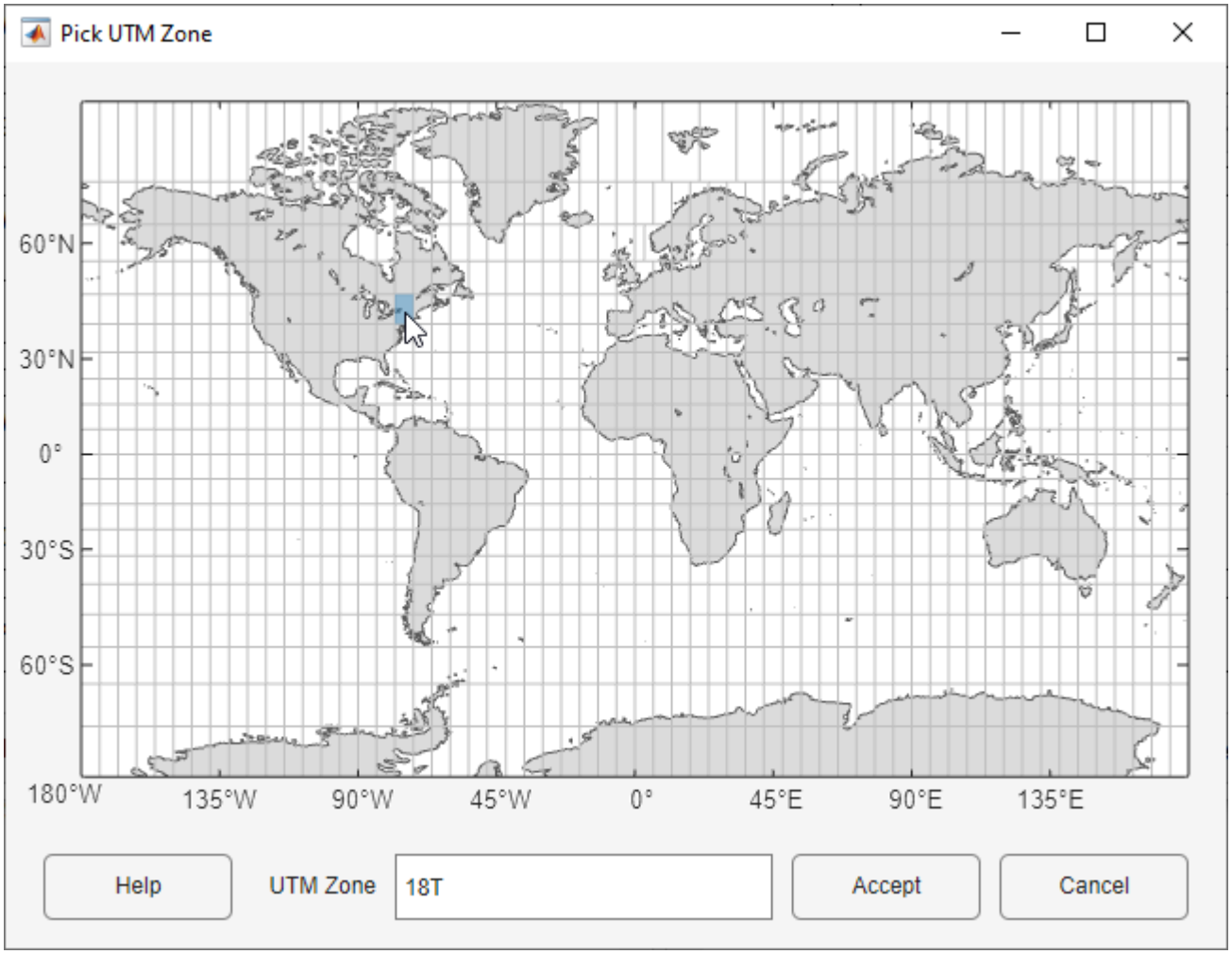
1. You can **Shift**+click in the window of an `axesm`-based map, or type `axesmui` to display the projection control panel. Here you start from scratch:

   ```
   figure
   axesm utm
   axesmui
   ```

   The **Map Projection** field is set to `cyln: Universal Transverse Mercator (UTM)`.

   Note For UTM and UPS maps, the **Aspect** field is set to `normal` and cannot be changed. If you attempt to specify `transverse`, an error results.

2. Click the **Zone** button to open the `utmzoneui` panel. Click the map near your area of interest to pick the zone:

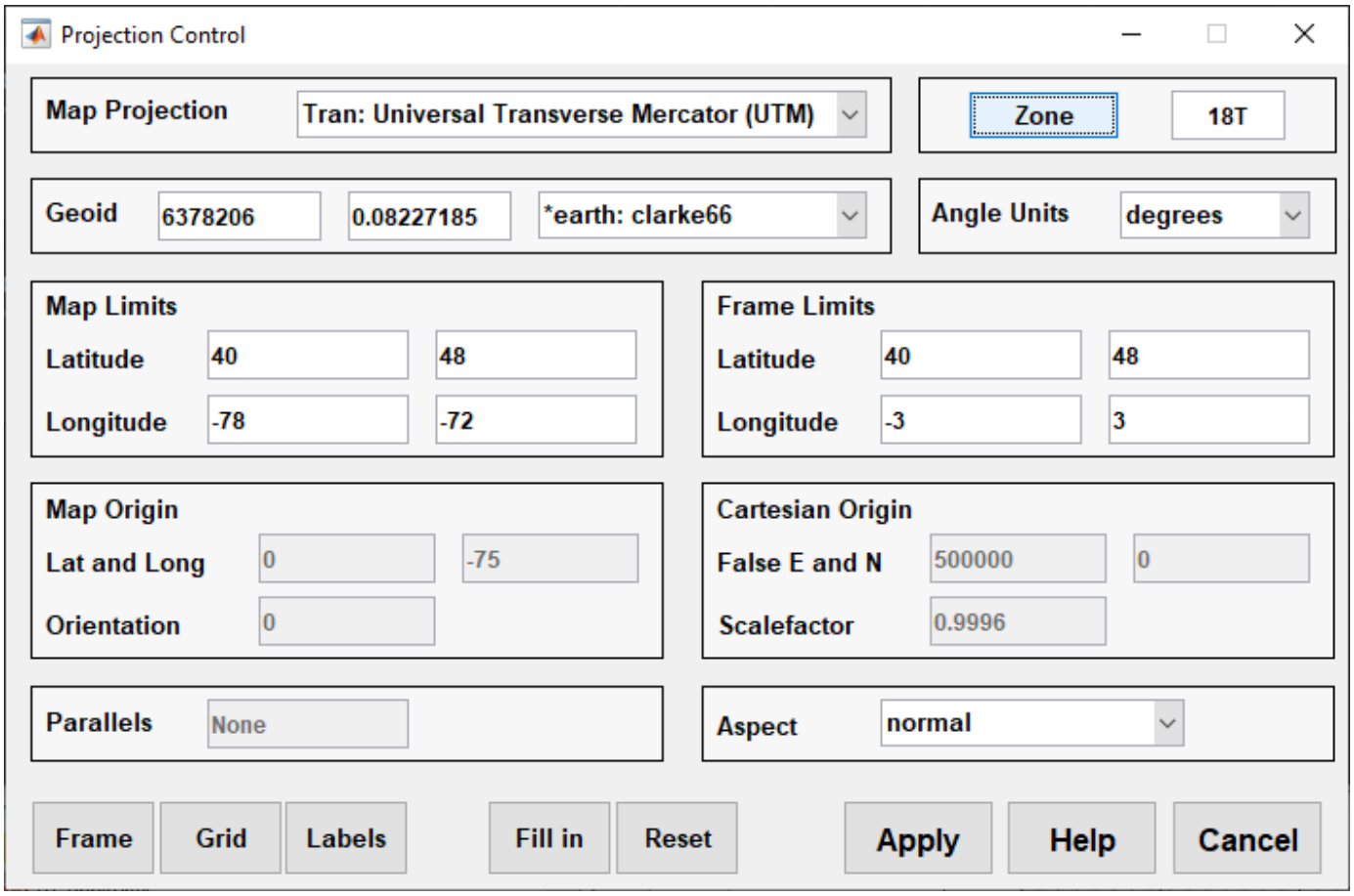


Note that while you can open the `utmzoneui` control panel from the command line, you then have to manually update the figure with the zone name it returns with a `setm` command:

```
setm(gca,'zone',ans)
```

3 Click the **Accept** button.

The `utmzoneui` panel closes, and the `zone` field is set to the one you picked. The map limits are updated accordingly, and the geoid parameters are automatically set to an appropriate ellipsoid definition for that zone. You can override the default choice by selecting another ellipsoid from the list or by typing the parameters in the **Geoid** field.

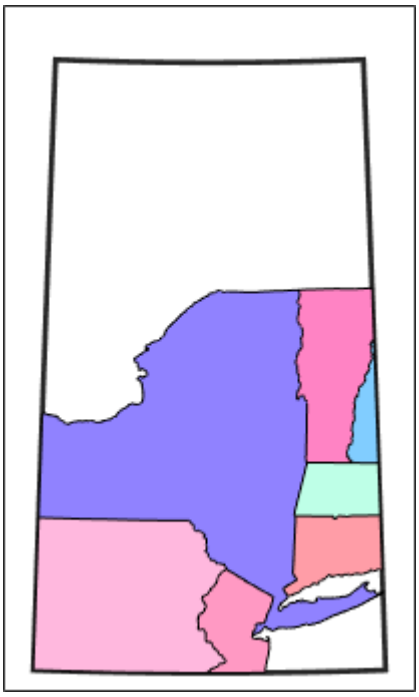


4 Click **Apply** to close the projection control panel.

The projection is then ready for projection calculations or map display commands.

5 Now view a choropleth base map from the `usstatehi` shapefile for the area within the zone that you just selected:

```
states = readgeotable("usastatehi.shp");
framem
faceColors = makesymbolspec("Polygon",...
    {"INDEX",[1 numel(states)],"FaceColor",polcmap(numel(states))});
geoshow(states,"DisplayType","polygon","SymbolSpec", faceColors)
```



What you see depends on the zone you selected. The preceding display is for zone 18T, which contains portions of New England and the Middle Atlantic states.

You can also calculate projected UTM grid coordinates from latitudes and longitudes:

```
[latlim, lonlim] = utmzone('15S')
```

```
latlim =

    32    40


lonlim =

   -96   -90
```

```
mstruct = gcm;
[x,y] = projfwd(mstruct, latlim, lonlim)
```

```
x =

   1.0e+06 *

   -1.5029   -0.7829


y =

   1.0e+06 *

    3.7403    4.5369
```

Work in UTM Without a Displayed Map

You can set up UTM to calculate coordinates without generating a map display, using the `defaultm` function. The `utmzone` and `utmgeoid` functions help you select a zone and an appropriate ellipsoid. In this example, you generate UTM coordinate data for a location in New York City, using that point to define the projection itself.

Define a location in New York City. Obtain the UTM zone for this point.

```
lat = 40.7;
lon = -74.0;
z = utmzone(lat,lon)
```

```
z =
'18T'
```

Get the suggested ellipsoid vector and name for this zone.

```
[ellipsoid,estr] = utmgeoid(z)
```

```
ellipsoid = 1×2
10^6 ×

    6.3782    0.0000


estr =
'clarke66'
```

Set up the UTM coordinate system based on this information.

```
utmstruct = defaultm('utm');
utmstruct.zone = z;
utmstruct.geoid = ellipsoid;
utmstruct = defaultm(utmstruct);
```

Transform the coordinates, without a map display.

```
[x,y] = projfwd(utmstruct,lat,lon)
```

```
x =
5.8448e+05

y =
4.5057e+06
```

Compute the zone limits (latitude and longitude limits) for a specified zone by using the `utmzone` function. You can also call `utmzone` recursively to obtain the limits of the UTM zone within which a point location falls.

```
utmzone('18T')
```

```
ans = 1×4

    40    48   -78   -72


[zonelats,zonelons] = utmzone(utmzone(40.7,-74.0))
```

```
zonelats = 1×2

    40    48


zonelons = 1×2

   -78   -72
```

Alternatively, set up a UTM coordinate system using a `projcrs` object. To create the object, specify an EPSG code. For information about EPSG codes, see the EPSG registry. Verify that the `projcrs` object has the correct name and ellipsoid. Then, transform the coordinates.

```
p = projcrs(26718);
p.Name

ans =
"NAD27 / UTM zone 18N"

p.GeographicCRS.Spheroid.Name

ans =
'Clarke 1866'

[xp,yp] = projfwd(p,lat,lon)

xp =
5.8448e+05

yp =
4.5057e+06
```

See Also

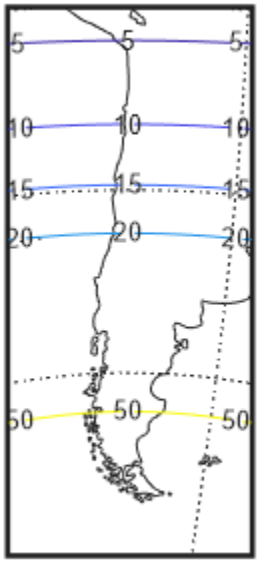
`projcrs` | `utmgeoid` | `utmzone` | `projfwd` | `defaultm`

Use the Transverse Aspect to Map Across UTM Zones

To display areas that extend across more than one UTM zone, use the Mercator projection in a transverse aspect. UTM is a zone-based coordinate system and is designed to be used like a map series, selecting from the appropriate sheet. While it is possible to extend one zone's coordinates into a neighboring zone's territory, this is not normally done. This example shows a transverse Mercator projection appropriate to Chile. In the example, note how the projection's line of zero distortion is aligned with the predominantly north-south axis of the country. Of course, you do not obtain coordinates in meters that would match those of a UTM projection, but the results will be nearly as accurate. To place the zero distortion line exactly on the midline of the country, use better estimates of the orientation vector's central meridian and orientation angle.

Set up an `axesm`-based map with a transverse aspect and display a map of Chile. Calculate the map distortion.

```
figure;
latlim = [-60 -15];
centralMeridian = -70;
width = 20;
axesm("mercator","Origin",[0 centralMeridian -90], ...
      "Flatlimit",[-width/2 width/2],"Flonlimit",sort(-latlim), ...
      "Aspect","transverse");
land = readgeotable("landareas.shp");
geoshow(land,"FaceColor","none")
framem
gridm
setm(gca,"plinefill",1000)
tightmap
mdistort scale
```



You might receive warnings about points from `landareas.shp` falling outside the valid projection region. You can ignore such warnings.

Map Projections and Distortions

Humans have known that the shape of the Earth resembles a sphere and not a flat surface since classical times, and possibly much earlier than that. If the world were indeed flat, cartography would be much simpler because map projections would be unnecessary.

A map projection is a procedure that flattens a curved surface such as the Earth onto a plane. Usually this is done through an intermediate surface such as a cylinder or a cone, which is then unwrapped to lie flat. Consequently, map projections are classified as cylindrical, conical, and azimuthal (a direct transformation of the surface of part of a spheroid to a circle). See “The Three Main Families of Map Projections” on page 9-15 for discussions and illustrations of how these transformations work. The toolbox can project both vector data and raster data.

Get a map projection by using a `projcrs` object. For example, get information about a raster data file by creating a `RasterInfo` object. Then, find the projection method by querying the `CoordinateReferenceSystem` property of the object.

```
info = georasterinfo('boston.tif');
info.CoordinateReferenceSystem.ProjectionMethod
```

```
ans =

    "Lambert Conic Conformal (2SP)"
```

Alternatively, Mapping Toolbox includes a library of map projections that you can control with `axesm` and `defaultm`. Some are suitable for showing the entire world, others for half of it, and some are only useful over small areas. For more information, see “Projection Distortions” on page 9-44. For a list of available projections, see “Summary and Guide to Projections” on page 9-2. If you want to use a projection that is not included in this list, create a `projcrs` object instead.

Project coordinates by using the `projfwd` function and unproject coordinates using the `projinv` function.

Use Inverse Projection to Recover Geographic Coordinates

When geospatial data has plane coordinates (i.e., it comes preprojected, as do many satellite images and municipal map data sets), it is usually possible to recover geographic coordinates if the projection parameters and datum are known. Using this information, you can perform an inverse projection, running the projection backward to solve for latitude and longitude. The toolbox can perform accurate inverse projections for any of its projection functions as long as the original projection parameters and reference ellipsoid (or spherical radius) are provided to it.

Note Converting a position given in latitude-longitude to its equivalent in a projected map coordinate system involves converting from units of angle to units of length. Likewise, unprojecting a point position changes its units from those of length to those of angle). Unit conversion functions such as `deg2km` and `km2deg` also convert coordinates between angles and lengths, but do not transform the space they inhabit. You cannot use them to project or unproject coordinate data.

Projection Distortions

All map projections introduce distortions compared to maps on globes. Distortions are inherent in flattening the sphere, and can take several forms:

- Areas — Relative size of objects (such as continents)
- Directions — Azimuths (angles between points and the poles)
- Distances — Relative separations of points (such as a set of cities)
- Shapes — Relative lengths and angles of intersection

Some classes of map projections maintain areas, and others preserve local shapes, distances, or directions. No projection, however, can preserve all these characteristics. Choosing a projection thus always requires compromising accuracy in some way, and that is one reason why so many different map projections have been developed. For any given projection, however, the smaller the area being mapped, the less distortion it introduces if properly centered. Mapping Toolbox tools help you to quantify and visualize projection distortions.

References

[1] Snyder, J. P. "Map Projections – A working manual." *U.S. Geological Survey Professional Paper 1395*. Washington, D.C.: U.S. Government Printing Office, 1987. doi:10.3133/pp1395

[2] Maling, D. H. *Coordinate Systems and Map Projections*. 2nd ed. New York: Pergamon Press, 1992.

[3] Snyder, J. P., and P. M. Voxland. "An album of map projections." *U.S. Geological Survey Professional Paper 1453*. Washington, D.C.; U.S. Government Printing Office, 1989. doi:10.3133/pp1453

[4] Snyder, J. P. *Flattening the Earth – 2000 Years of Map Projections*. Chicago, IL: University of Chicago Press, 1993.

See Also

More About

- “Quantitative Properties of Map Projections” on page 9-19
- “Projection Parameters” on page 9-21

Visualize Spatial Error Using Tissot Indicatrices

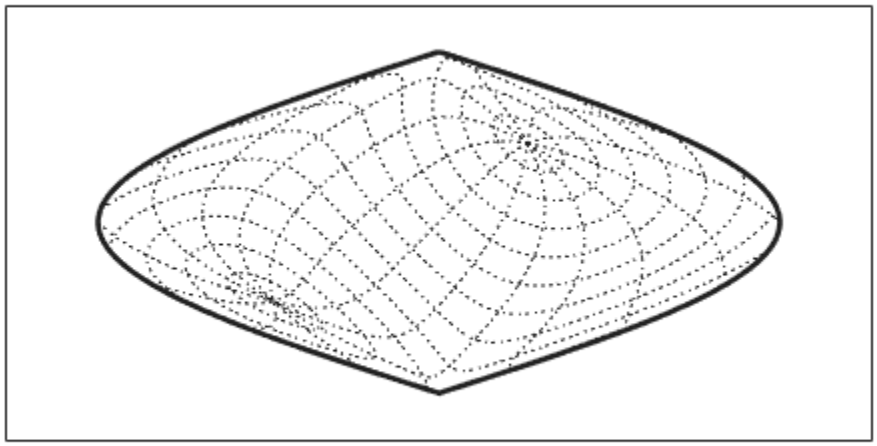
A standard method of visualizing the distortions introduced by the map projection is to display small circles at regular intervals across the globe. After projection, the small circles appear as ellipses of various sizes, elongations, and orientations. The sizes and shapes of the ellipses reflect the projection distortions. Conformal projections have circular ellipses, while equal-area projections have ellipses of the same area. This method was invented by Nicolas Tissot in the 19th century, and the ellipses are called *Tissot indicatrices* in his honor. The measure is a tensor function of location that varies from place to place, and reflects the fact that, unless a map is conformal, map scale is different in every direction at a location.

Visualize Projection Distortions Using Tissot Indicatrices

This example shows how to add Tissot indicatrices to a map display.

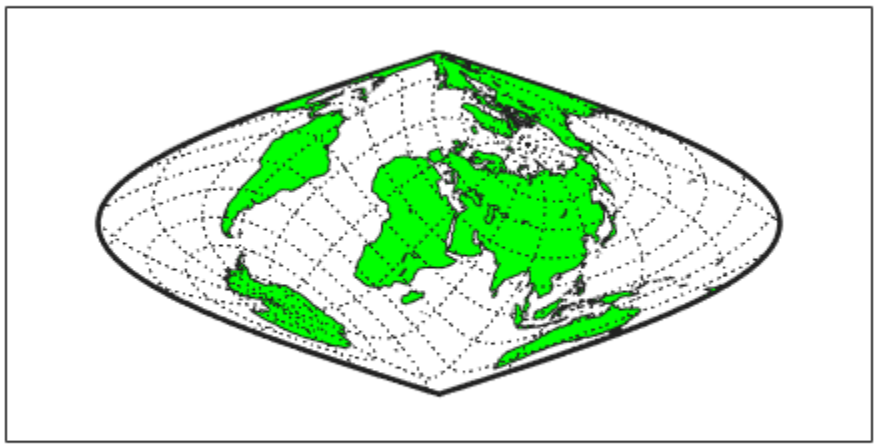
Set up a Sinusoidal projection in a skewed aspect, plotting the graticule.

```
figure
axesm sinusoid
gridm on
framem on
setm(gca,'Origin',[20 30 45])
```



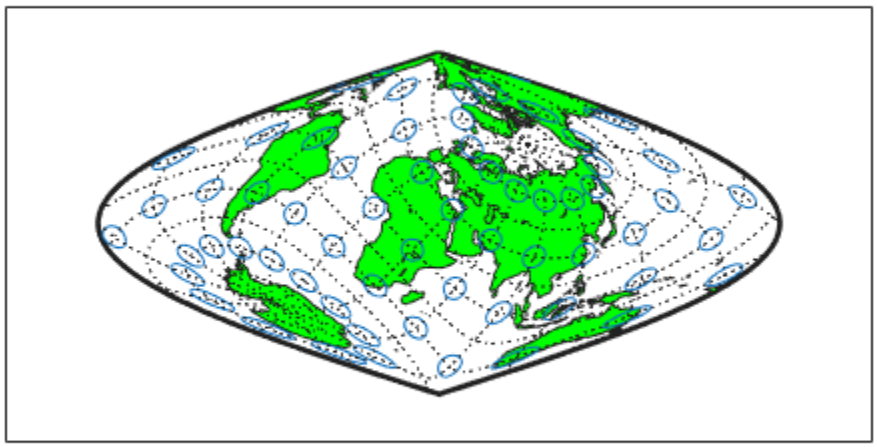
Load the coast data set and plot it as green patches.

```
load coastlines
patchm(coastlat,coastlon,'g')
```



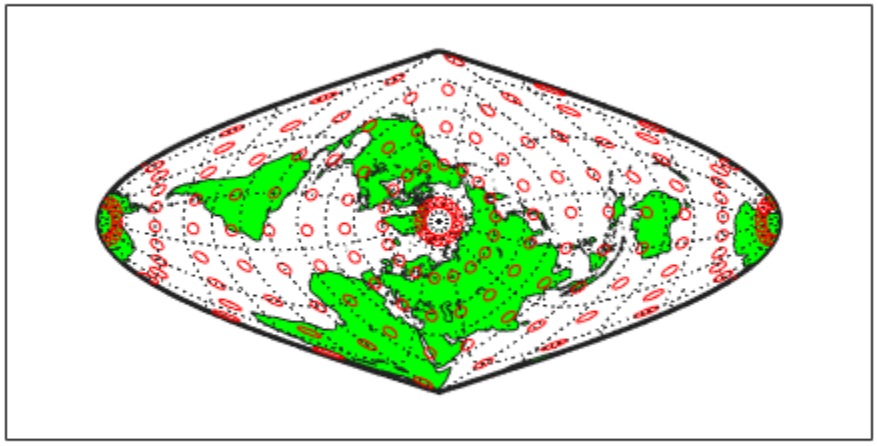
Plot the default Tissot diagram. Notice that the circles vary considerably in shape. This indicates that the Sinusoidal projection is not conformal. Despite the distortions, however the circles all cover equal amounts of area on the map because the projection has the equal-area property. Default Tissot diagrams are drawn with blue unfilled 100-point circles spaced 30 degrees apart in both directions. The default circle radius is 1/10 of the current radius of the reference ellipsoid (by default that radius is 1).

```
tissot
```



Clear the Tissot diagram, rotate the projection to a polar aspect, and plot a new Tissot diagram using circles paced 20 degrees apart, half as big as before, drawn with 20 points, and drawn in red. In the result, note that the circles are drawn faster because fewer points are computed for each one. Also note that the distortions are still smallest close to the map origin, and still greatest near the map frame.

```
clmo tissot
setm(gca,'Origin',[90 0 45])
tissot([20 20 .05 20],'Color','r')
```



See Also

`distortcalc` | `mdistort` | `tissot`

More About

- “Quantitative Properties of Map Projections” on page 9-19

Quantify Map Distortions at Point Locations

The `tissot` and `mdistort` functions provide synoptic visual overviews of different forms of map projection error. Sometimes, however, you need numerical estimates of error at specific locations in order to quantify or correct for map distortions. This is useful, for example, if you are sampling environmental data on a uniform basis across a map, and want to know precisely how much area is associated with each sample point, a statistic that will vary by location and be projection dependent. Once you have this information, you can adjust environmental density and other statistics you collect for areal variations induced by the map projection.

A Mapping Toolbox function returns location-specific map error statistics from the current projection or an `mstruct`. The `distortcalc` function computes the same distortion statistics as `mdistort` does, but for specified locations provided as arguments. You provide the latitude-longitude locations one at a time or in vectors. The general form is

```
[areascale,angdef,maxscale,minscale,merscale,parscale] = ...
    distortcalc(mstruct,lat,long)
```

However, if you are evaluating the current map figure, omit the `mstruct`. You need not specify any return values following the last one of interest to you.

Use distortcalc to Determine Map Projection Geometric Distortions

The following exercise uses `distortcalc` to compute the maximum area distortion for a map of Argentina from the land areas data set.

1 Read the North and South America polygon:

```
Americas = shaperead('landareas.shp','UseGeoCoords',true, ...
    'Selector', {@(name) ...
    strcmpi(name,{'north and south america'}),'Name'});
```

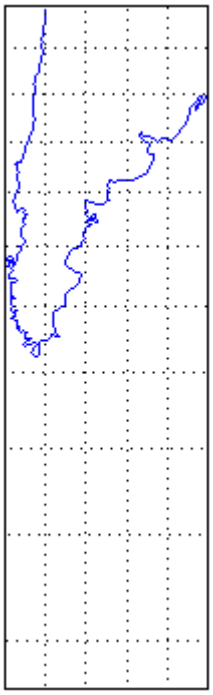
2 Set the spatial extent (map limits) to contain the southern part of South America and also include an area closer to the South Pole:

```
mlatlim = [-72.0 -20.0];
mlonlim = [-75.0 -50.0];
[alat, alon] = maptriml([Americas.Lat], ...
    [Americas.Lon], mlatlim, mlonlim);
```

3 Create a Mercator cylindrical conformal projection using these limits, specify a five-degree graticule, and then plot the outline for reference:

```
figure;
axesm('MapProjection','mercator','grid','on', ...
    'MapLatLimit',mlatlim,'MapLonLimit',mlonlim,...
    'MLineLocation',5, 'PLineLocation',5)
plotm(alat,alon,'b')
```

The map looks like this:



4 Sample every tenth point of the patch outline for analysis:

```
alats = alat(1:10:numel(alat));
alons = alon(1:10:numel(alat));
```

5 Compute the area distortions (the first value returned by `distortcalc`) at the sample points:

```
adistort = distortcalc(alats, alons);
```

6 Find the range of area distortion across Argentina (percent of a unit area on, in this case, the equator):

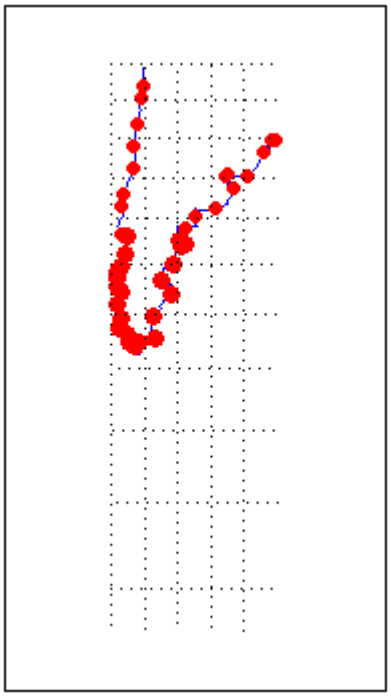
```
adistortmm = [min(adistort) max(adistort)]

adistortmm =
    1.1790    2.7716
```

As Argentina occupies mid southern latitudes, its area on a Mercator map is overstated, and the errors vary noticeably from north to south.

7 Remove any `NaN`s from the coordinate arrays and plot symbols to represent the relative distortions as proportional circles, using `scatterm`:

```
nanIndex = isnan(adistort);
alats(nanIndex) = [];
alons(nanIndex) = [];
adistort(nanIndex)  = [];
scatterm(alats,alons,20*adistort,'red','filled')
```

The resulting map is shown below:



8 The degree of area overstatement would be considerably larger if it extended farther toward the pole. To see how much larger, get the area distortion for 50°S, 60°S, and 70°S:

```
a=distortcalc(-50,-60)

a =
       2.4203

a=distortcalc(-60,-60)

a =
            4

>> a=distortcalc(-70,-60)

a =
       8.5485
```

Note You can only use `distortcalc` to query locations that are within the current map frame or `mstruct` limits. Outside points yield `NaN` as a result.

9 Using this technique, you can write a simple script that lets you query a map repeatedly to determine distortion at any desired location. You can select locations with the graphic cursor using `inputm`. For example,

```
[plat plon] = inputm(1)

plat =
      -62.225
plon =
      -72.301
```

```
>> a=distortcalc(plat,plon)

a =
       4.6048
```

Naturally the answer you get will vary depending on what point you pick. Using this technique, you can write a simple script that lets you query a map repeatedly to determine any distortion statistic at any desired location.

Try changing the map projection or even the orientation vector to see how the choice of projection affects map distortion. For further information, see the reference page for `distortcalc`.

Projection Aspect

A map projection's *aspect* is its orientation on the page or display screen. If north or south is straight up, the aspect is said to be *equatorial*; for most projections this is the *normal* aspect. When the central axis of the developable surface is oriented east-west, the projection's aspect is *transverse*. Projections centered on the North Pole or the South Pole have a *polar* aspect, regardless of what meridian is up. All other orientations have an *oblique* aspect. So far, the examples and discussions of map displays have focused on the *normal* aspect, by far the most commonly used. This section discusses the use of *transverse*, *oblique*, and *skew-oblique* aspects. For an example, see “Control the Map Projection Aspect with an Orientation Vector” on page 9-55.

Projection aspect is primarily of interest in the display of maps. However, this section also discusses how the idea of projection aspect as a coordinate system transformation can be applied to map variables for analytical purposes.

Note The projection aspect discussed in this section is different from the `Aspect` property of `axesm`-based maps. The `Aspect` property of `axesm`-based maps controls the orientation of the figure axes. For instance, if a map is in a `normal` setting with a *landscape* orientation, a switch to a `transverse` aspect rotates the axes by 90°, resulting in a *portrait* orientation. To display a map in the transverse aspect, combine the `transverse` aspect property with a -90° skew angle. The skew angle is the last element of the `Origin` parameter. For example, a `[0 0 -90]` vector would produce a transverse map.

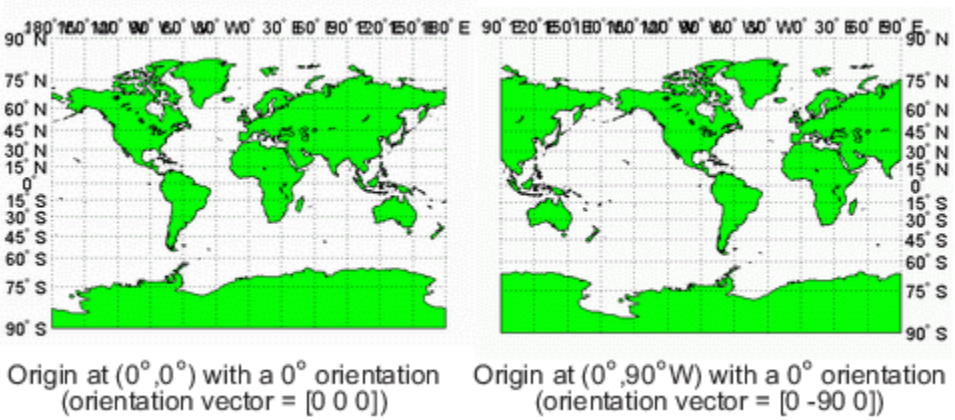
The Orientation Vector

The `Origin` property of `axesm`-based maps is a vector describing the geometry of the displayed projection. This property is called an *orientation vector* (prior versions called it the *origin vector*). The vector takes this form:

```
orientvec = [latitude longitude orientation]
```

The latitude and longitude represent the geographic coordinates of the center point of the display from which the projection is calculated. The orientation refers to the clockwise angle from *straight up* at which the North Pole points from this center point. The default orientation vector is `[0 0 0]`; that is, the projection is centered on the geographic point (0°,0°) and the North Pole is *straight up* from this point. Such a display is in a *normal* aspect. Changes to only the longitude value of the orientation vector do not change the aspect; thus, a normal aspect is one centered on the Equator in latitude with an orientation of 0°.

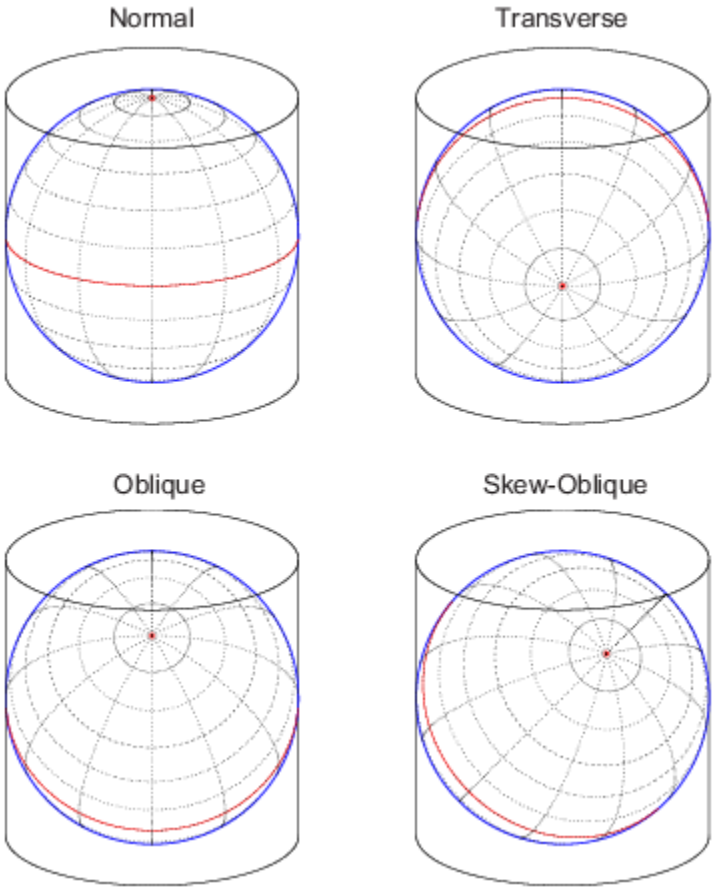
Both of these Miller projections have normal aspects, despite having different orientation vectors:



This makes sense if you think about a simple, true cylindrical projection. This is the projection of the globe onto a cylinder wrapped around it. For normal aspects, this cylinder is tangent to the globe at the Equator, and changing the origin longitude simply corresponds to rotating the sphere about the longitudinal axis of the cylinder. If you continue with the wrapped-cylinder model, you can understand the other aspects as well.

Following this description, a *transverse* projection can be thought of as a cylinder wrapped around the globe tangent at the poles and along a meridian and its antipodal meridian. Finally, when such a cylinder is tangent along any great circle other than a meridian, the result is an *oblique* projection.

Here are diagrams of the four cylindrical map orientations, or aspects:



Of course, few projections are true cylindrical projections, but the concept of the wrapped cylinder is nonetheless a convenient way to describe aspect.

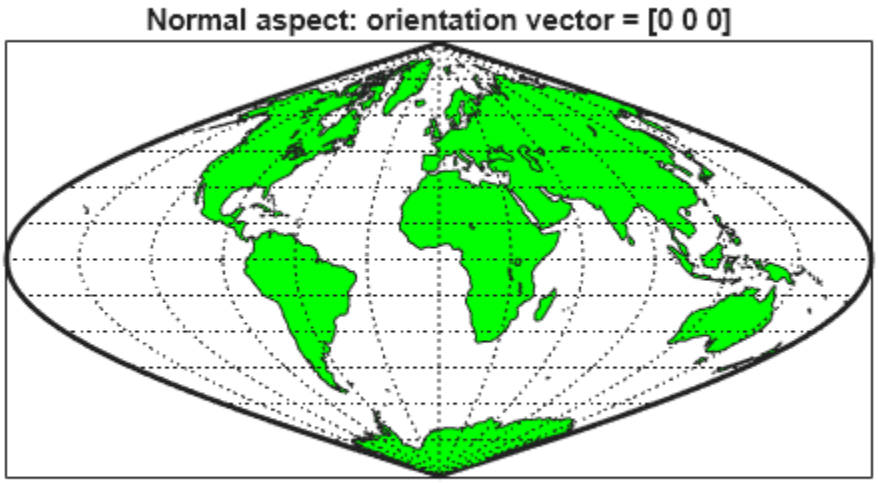
For an example that uses the orientation vector, see “Control the Map Projection Aspect with an Orientation Vector” on page 9-55.

Control the Map Projection Aspect with an Orientation Vector

The best way to gain an understanding of projection aspect is to experiment with orientation vectors. The following example uses a pseudocylindrical projection, the sinusoidal.

Create a default `axesm`-based map in a sinusoidal projection, turn on the graticule, and display the coast data set as filled polygons. The continents and graticule appear in *normal* aspect.

```
figure
axesm sinusoid
framem on
gridm on
tightmap tight
load coastlines
patchm(coastlat,coastlon,'g')
title('Normal aspect: orientation vector = [0 0 0]')
```



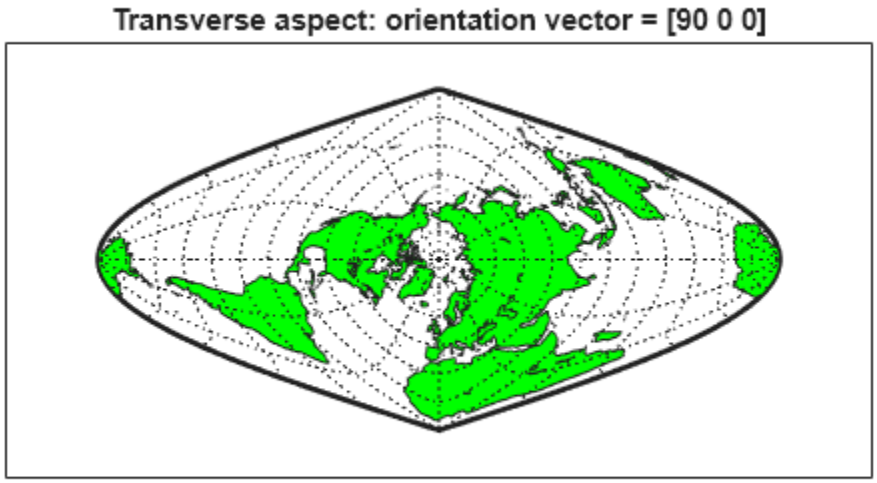
Inspect the orientation vector from the `axesm`-based map. By default, the origin is set at (0°E, 0°N), oriented 0° from vertical.

```
getm(gca,'Origin')
```

```
ans = 1×3

     0     0     0
```

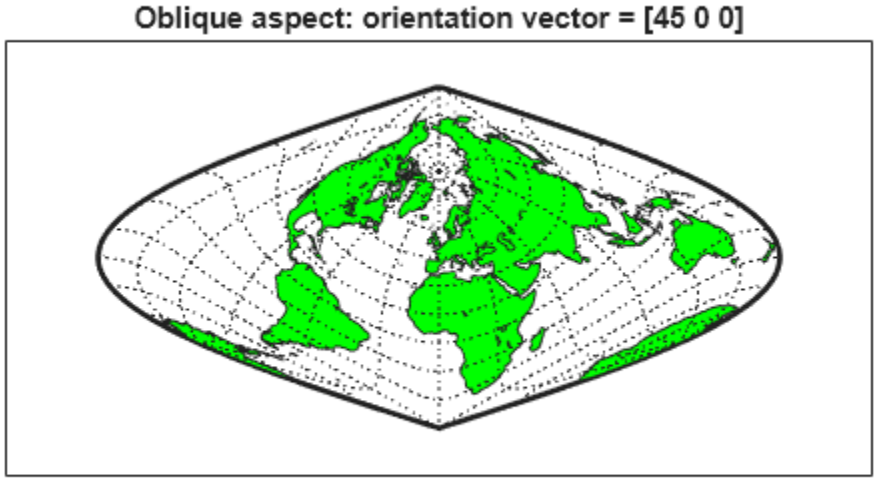
In the normal aspect, the North Pole is at the top of the image. To create a *transverse* aspect, imagine pulling the North Pole down to the center of the display, which was originally occupied by the point (0°,0°). Do this by setting the first element of `Origin` parameter to a latitude of 90°N. The shape of the frame is unaffected. This is still a sinusoidal projection.

```
setm(gca,'Origin',[90 0 0])
title('Transverse aspect: orientation vector = [90 0 0]')
```



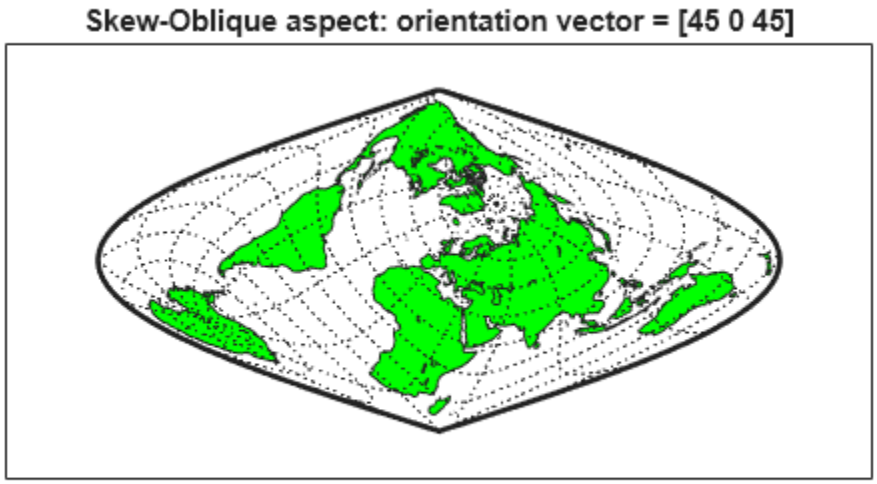
The normal and transverse aspects can be thought of as limiting conditions. Anything else is an *oblique* aspect. Conceptually, if you push the North Pole halfway back to its original position, that is, to the position originally occupied by the point (45°N, 0°E) in the normal aspect, the result is a simple oblique aspect. You can think of this as pulling the new origin (45°N, 0°) to the center of the image, the place that (0°,0°) occupied in the normal aspect.

```
setm(gca,'Origin',[45 0 0])
title('Oblique aspect: orientation vector = [45 0 0]')
```



The previous examples of projection aspect kept the aspect orientation at 0°. If you alter the orientation, an oblique aspect becomes a *skew-oblique* orientation. Imagine the previous example with an orientation of 45°. Think of this as pulling the new origin (45°N,0°E), down to the center of the projection and then rotating the projection until the North Pole lies at an angle of 45° clockwise from straight up with respect to the new origin. As in the previous example, the location (45°N,0°E) still occupies the center of the map.

```
setm(gca,'Origin',[45 0 45])
title('Skew-Oblique aspect: orientation vector = [45 0 45]')
```



The base projection can be thought of as a standard coordinate system, and the normal aspect conforms to it. The features of a projection are maintained in any aspect, *relative to the base projection*. As the preceding illustrations show, the outline (frame) does not change. Nondirectional projection characteristics also do not change. For example, the sinusoidal projection is equal-area, no matter what its aspect. Directional characteristics must be considered carefully, however. In the normal aspect of the sinusoidal projection, scale is true along every parallel and the central meridian. This is not the case for the skew-oblique aspect; however, scale is true along the paths of the transformed parallels and meridian.

Any projection can be viewed in alternate aspects and this can often be quite useful. For example, the transverse aspect of the Mercator projection is widely used in cartography, especially for mapping regions with predominantly north-south extent. One candidate for such handling might be Chile. Oblique Mercator projections might be used to map long regions that run neither north and south nor east and west, such as New Zealand.

Rotational Transformations on the Globe

In “The Orientation Vector” on page 9-53, you explored the concept of altering the aspect of a map projection in terms of pushing the North Pole to new locations. Another way to think about this is to redefine the coordinate system, and then to compute a normal aspect projection based on the new system. For example, you might redefine a spherical coordinate system so that your home town occupies the origin. If you calculated a map projection in a normal aspect with respect to this *transformed* coordinate system, the resulting display would look like an oblique aspect of the *true* coordinate system of latitudes and longitudes.

This transformation of coordinate systems can be useful independent of map displays. If you transform the coordinate system so that your home town is the new *North Pole*, then the transformed coordinates of all other points will provide interesting information.

Note The types of coordinate transformations described here are appropriate for the spherical case only. Attempts to perform them on an ellipsoid will produce incorrect answers on the order of several to tens of meters.

When you place your home town at a pole, the spherical distance of each point from your hometown becomes 90° minus its transformed latitude (also known as a *colatitude*). The point antipodal to your town would become the *South Pole*, at -90°. Its distance from your hometown is 90°-(-90°), or 180°, as expected. Points 90° distant from your hometown all have a transformed latitude of 0°, and thus make up the transformed *equator*. Transformed longitudes correspond to their respective great circle azimuths from your home town.

Reorient Vector Data with rotatem

The `rotatem` function uses an orientation vector to transform latitudes and longitudes into a new coordinate system. The orientation vector can be produced by the `newpole` or `putpole` functions, or can be specified manually.

As an example of transforming a coordinate system, suppose you live in Midland, Texas, at (32°N,102°W). You have a brother in Tulsa (36.2°N,96°W) and a sister in New Orleans (30°N,90°W).

1. Define the three locations:

```
midl_lat = 32;   midl_lon = -102;
tuls_lat = 36.2; tuls_lon = -96;
newo_lat = 30;   newo_lon = -90;
```

2. Use the `distance` function to determine great circle distances and azimuths of Tulsa and New Orleans from Midland:

```
[dist2tuls az2tuls] = distance(midl_lat,midl_lon,...
                               tuls_lat,tuls_lon)

dist2tuls =
   6.5032

az2tuls =
  48.1386

[dist2neworl az2neworl] = distance(midl_lat,midl_lon,...
```

```
                                    newo_lat,newo_lon)

dist2neworl =
  10.4727

az2neworl =
  97.8644
```

Tulsa is about 6.5 degrees distant, New Orleans about 10.5 degrees distant.

3. Compute the absolute difference in azimuth, a fact you will use later.

```
azdif = abs(az2tuls-az2neworl)

azdif =
   49.7258
```

4. Today, you feel on top of the world, so make Midland, Texas, the *north pole* of a transformed coordinate system. To do this, first determine the origin required to put Midland at the pole using `newpole`:

```
origin = newpole(midl_lat,midl_lon)

origin =
    58    78     0
```

The origin of the new coordinate system is (58°N, 78°E). Midland is now at a *new latitude* of 90°.

5. Determine the transformed coordinates of Tulsa and New Orleans using the `rotatem` command. Because its units default to radians, be sure to include the `degrees` keyword:

```
[tuls_lat1,tuls_lon1] = rotatem(tuls_lat,tuls_lon,...
                                origin,'forward','degrees')

tuls_lat1 =
    83.4968
tuls_lon1 =
    -48.1386

[newo_lat1,newo_lon1] = rotatem(newo_lat,newo_lon,...
                                origin,'forward','degrees')

newo_lat1 =
    79.5273
newo_lon1 =
    -97.8644
```

6. Show that the new colatitudes of Tulsa and New Orleans equal their distances from Midland computed in step 2 above:

```
tuls_colat1 = 90-tuls_lat1

tuls_colat1 =
    6.5032

newo_colat1 = 90-newo_lat1

newo_colat1 =
   10.4727
```

7 Recall from step 4 that the absolute difference in the azimuths of the two cities from Midland was 49.7258°. Verify that this equals the difference in their new longitudes:

```
tuls_lon1-newo_lon1

ans =
   49.7258
```

You might note small numerical differences in the results (on the order of 10^{-6}), due to round-off error and trigonometric functions.

For further information, see the reference pages for `rotatem`, `newpole`, `putpole`, `neworig`, and `org2pol`.

Reorient Gridded Data

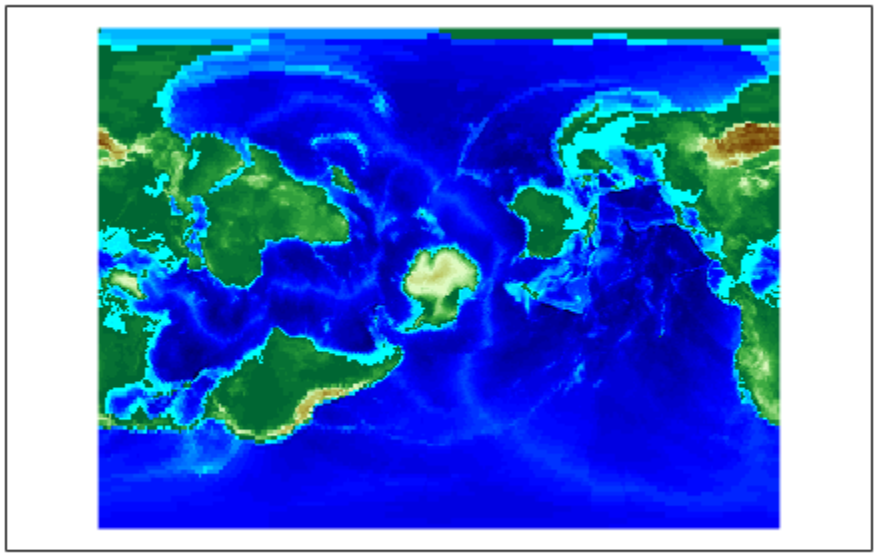
This example shows how to transform a regular data grid into a new one with its data rearranged to correspond to a new coordinate system using the `neworig` function. You can transform coordinate systems of data grids as well as vector data. When regular data grids are manipulated in this manner, distance and azimuth calculations with the map variable become row and column operations.

Load elevation raster data and a geographic cells reference object. Transform the data set to a new coordinate system in which a point in Sri Lanka is the north pole. Reorient the data grid by using the `neworig` function. Note that the result, `[Z,lat,lon]`, is a geolocated data grid, not a regular data grid like the original data.

```
load topo60c
origin = newpole(7,80);
[Z,lat,lon] = neworig(topo60c,topo60cR,origin);
```

Display the new map, in normal aspect, as its orientation vector shows. Note that every cell in the first row of the new grid is 0 to 1 degrees distant from the point new origin. Every cell in its second row is 1 to 2 degrees distant, and so on. In addition, every cell in a particular column has the same great circle azimuth from the new origin.

```
axesm miller
lat = linspace(-90,90,90);
lon = linspace(-180,180,180);
surfm(lat,lon,Z);
demcmap(topo60c)
```



```
mstruct = getm(gca);
mstruct.origin
```

```
ans = 1×3

     0     0     0
```

Map Projections — Alphabetical List

aitoff

Aitoff projection

Classification

Modified Azimuthal

Identifier

`aitoff`

Graticule

Meridians: Central meridian is a straight line half the length of the Equator. Other meridians are complex curves, equally spaced along the Equator, and concave toward the central meridian.

Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is neither conformal nor equal area. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

Parallels

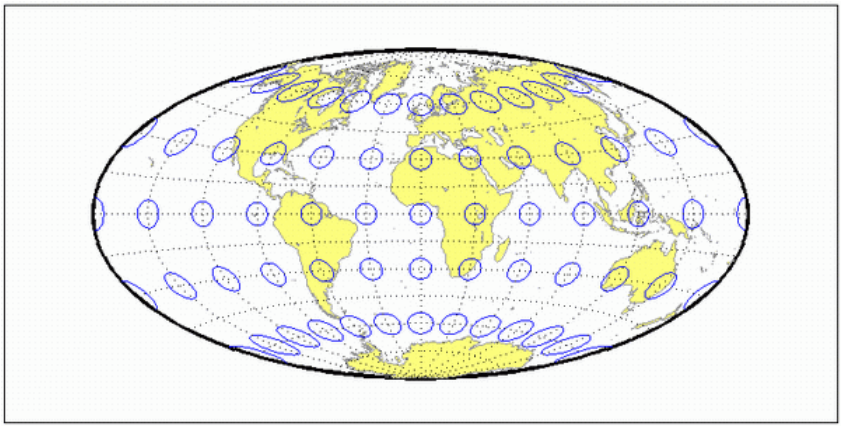
There is no standard parallel for this projection.

Remarks

- This projection was created by David Aitoff in 1889. It is a modification of the Equidistant Azimuthal projection. The Aitoff projection inspired the similar Hammer projection, which is equal area.
- This implementation of the Aitoff projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Aitoff projection using the ESRI authority code 54043. For example: `projcrs(54043,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('aitoff', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eqaconic

Albers Equal-Area Conic Projection

Classification

Conic

Identifier

`eqaconic`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the point of convergence. Spacing of parallels decreases away from the central latitudes.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

This is an equal-area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal-area. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

Remarks

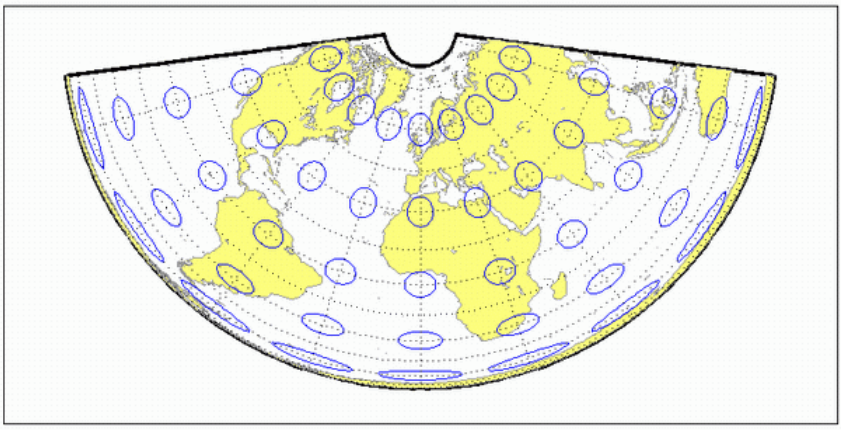
This projection was presented by Heinrich Christian Albers in 1805.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqaconic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqaconicstd` on page 10-6

Version History

Introduced before R2006a

eqaconicstd

Albers Equal-Area Conic Projection — Standard

Classification

Conic

Identifier

`eqaconicstd`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the point of convergence. Spacing of parallels decreases away from the central latitudes.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

This function implements the Albers Equal Area Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `eqaconic` on page 10-4 for an alternative implementation based on rotating the authalic sphere.

This is an equal area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal area. The projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

Parallels

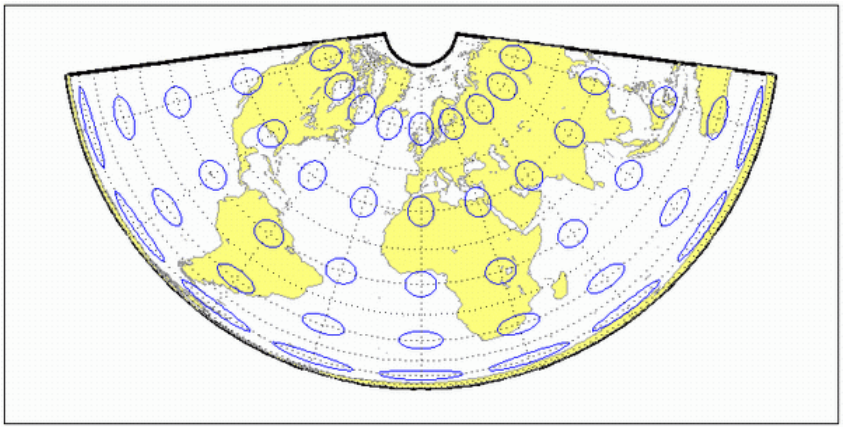
The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

Remarks

- This projection was presented by Heinrich Christian Albers in 1805 and it is also known as a Conical Orthomorphic projection. The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Lambert Equal Area Conic projection is the result. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Cylindrical Equal Area Projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other cylindrical equal area projection is the result.
- Mapping Toolbox uses a different implementation of the standard Albers equal-area conic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqaconicstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See also

`eqaconic` on page 10-4

Version History

Introduced before R2006a

apianus

Apianus II Projection

Classification

Pseudocylindrical

Identifier

`apianus`

Graticule

Meridians: Equally spaced elliptical curves converging at the poles.

Parallels: Equally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

Scale is constant along any parallel or pair of parallels equidistant from the Equator, as well as along the central meridian. The Equator is free of angular distortion. This projection is not equal-area, equidistant, or conformal.

Parallels

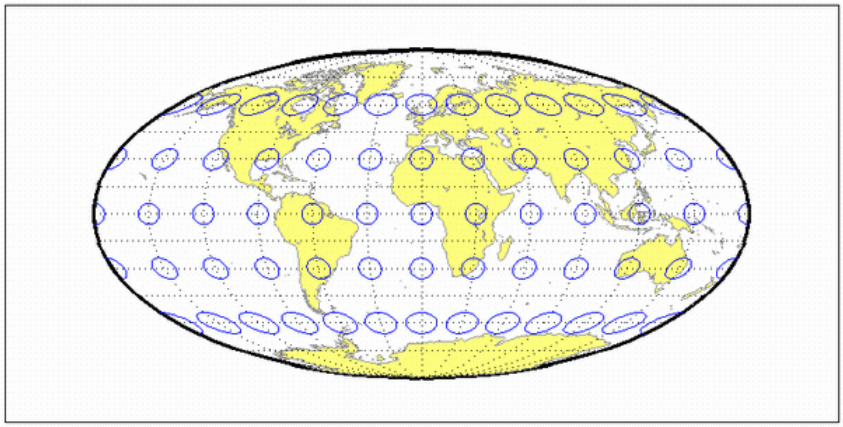
There is no standard parallel for this projection.

Remarks

- This projection was first described in 1524 by Peter Apian (or Bienewitz).
- This implementation of the Apianus II projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('apianus', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

balthsrt

Balthasart Cylindrical Projection

Classification

Cylindrical

Identifier

`balthsrt`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 50º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Standard Parallels

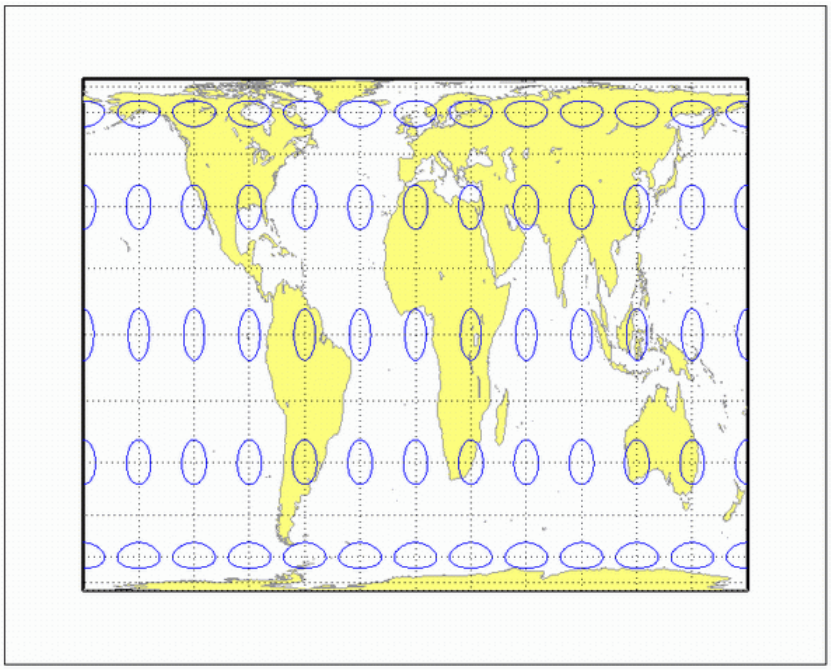
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 50º.

Remarks

The Balthasart Cylindrical projection was presented in 1935 and is a special form of the Equal-Area Cylindrical projection secant at 50ºN and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('balthsrt', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

behrmann

Behrmann Cylindrical Projection

Classification

Cylindrical

Identifier

`behrmann`

Graticule

Meridians: Equally spaced straight parallel lines 0.42 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 30º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

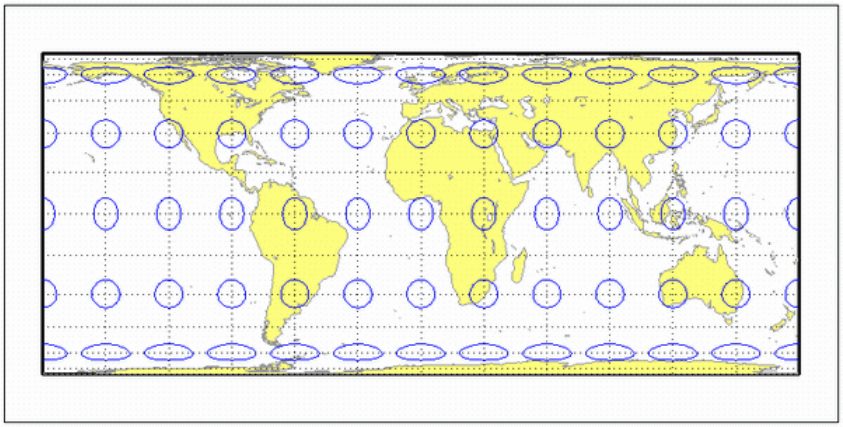
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 30º.

Remarks

This projection is named for Walter Behrmann, who presented it in 1910 and is a special form of the Equal-Area Cylindrical projection secant at 30ºN and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('behrmann', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

bsam

Bolshoi Sovietskii Atlas Mira Projection

Classification

Cylindrical

Identifier

`bsam`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 30º parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

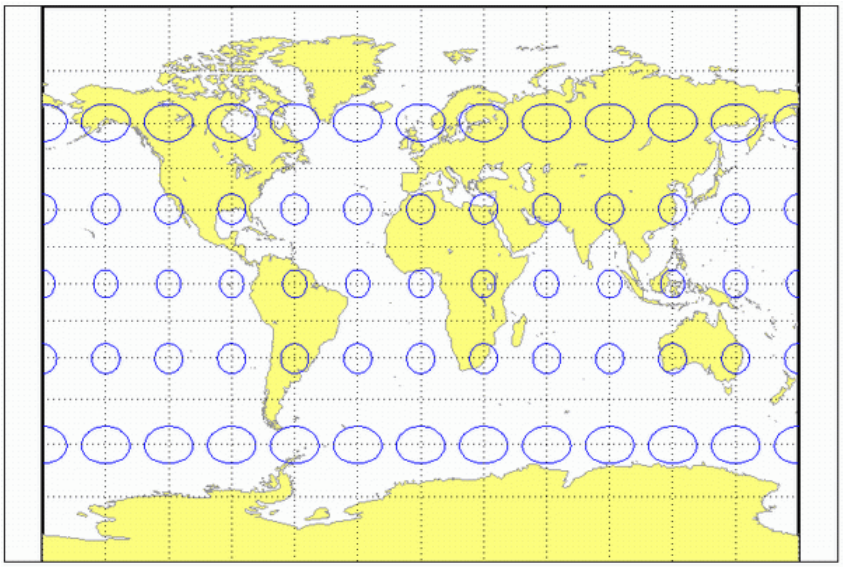
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 30º.

Remarks

- This projection was first described in 1937, when it was used for maps in the *Bolshoi Sovietskii Atlas Mira* (Great Soviet World Atlas). It is commonly abbreviated as the BSAM projection. It is a special form of the Braun Perspective Cylindrical projection secant at 30ºN and S.
- This implementation of the Bolshoi Sovietskii Atlas Mira projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('bsam', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

bonne

Bonne Projection

Classification

Pseudoconic

Identifier

bonne

Graticule

Central Meridian: A straight line.

Meridians: Complex curves connecting points equally spaced along each parallel and concave toward the central meridian.

Parallels: Concentric circular arcs spaced at true distances along the central meridian.

Poles: Points.

Symmetry: About the central meridian.

Features

This is an equal-area projection. The curvature of the standard parallel is identical to that on a cone tangent at that latitude. The central meridian and the central parallel are free of distortion. This projection is not conformal.

Parallels

This projection has one standard parallel, which is 30ºN by default. It has two interesting limiting forms. If a pole is employed as the standard parallel, a Werner projection results; if the Equator is used, a Sinusoidal projection results.

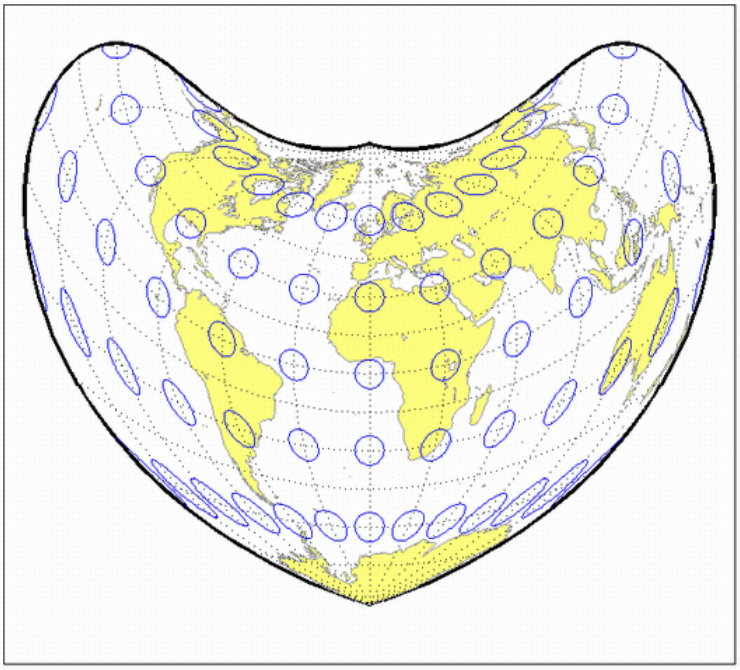
Remarks

This projection dates in a rudimentary form back to Claudius Ptolemy (about A.D. 100). It was further developed by Bernardus Sylvanus in 1511. It derives its name from its considerable use by Rigobert Bonne, especially in 1752.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('bonne', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

braun

Braun Perspective Cylindrical Projection

Classification

Cylindrical

Identifier

`braun`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at standard parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude may be chosen; the default is arbitrarily set to 0º.

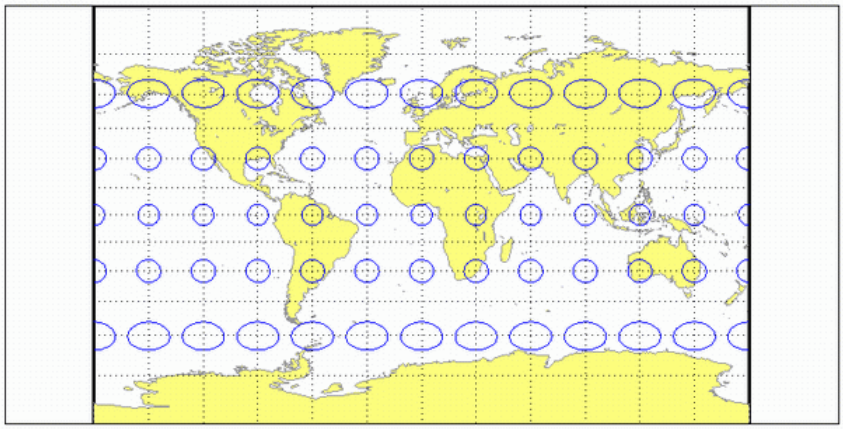
Remarks

- This projection was first described by Braun in 1867. It is less well known than the specific forms of it called the Gall Stereographic and the *Bolshoi Sovietskii Atlas Mira* projections.
- This implementation of the Braun perspective cylindrical projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('braun', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

breusing

Breusing Harmonic Mean Projection

Classification

Azimuthal

Identifier

`breusing`

Graticule

The graticule described is for the polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole.

Parallels: Unequally spaced circles centered on the central pole. The opposite hemisphere cannot be shown. Spacing increases (slightly) away from the central pole.

Poles: The central pole is a point, while the opposite pole cannot be shown.

Symmetry: About any meridian.

Features

This is a harmonic mean between a Stereographic and Lambert Equal-Area Azimuthal projection. It is not equal-area, equidistant, or conformal. There is no point at which scale is accurate in all directions. The primary feature of this projection is that it is minimum error—distortion is moderate throughout.

Parallels

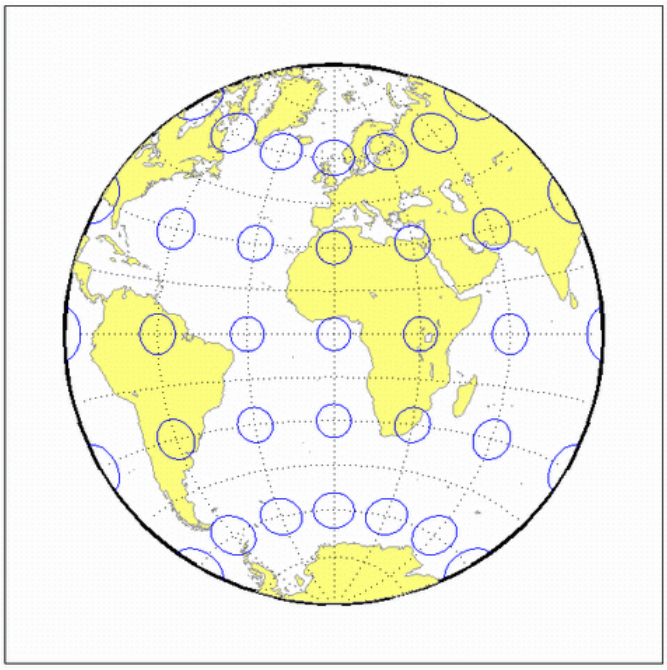
There are no standard parallels for azimuthal projections.

Remarks

- F. A. Arthur Breusing developed a geometric mean version of this projection in 1892. A. E. Young modified this to the harmonic mean version presented here in 1920. This projection is virtually indistinguishable from the Airy Minimum Error Azimuthal projection, presented by George Airy in 1861.
- This implementation of the Breusing harmonic mean projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('breusing', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

bries

Briesemeister Projection

Classification

Modified Azimuthal

Identifier

`bries`

Graticule

Meridians: Central meridian is straight. Other meridians are complex curves.

Parallels: Complex curves.

Poles: Points.

Symmetry: About the central meridian.

Features

This equal-area projection groups the continents about the center of the projection. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout.

Parallels

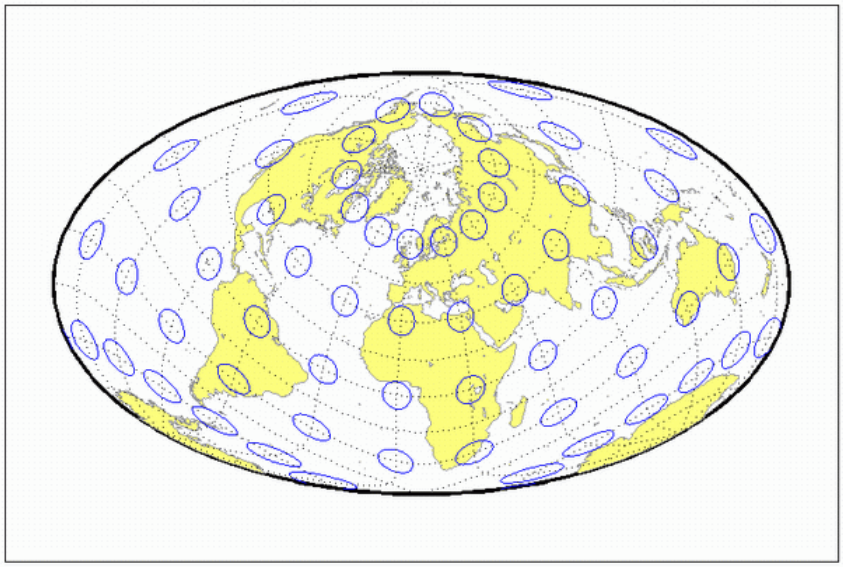
There is no standard parallel for this projection.

Remarks

- This projection was presented by William Briesemeister in 1953. It is an oblique Hammer projection with an axis ratio of 1.75 to 1, instead of 2 to 1.
- This implementation of the Briesemeister projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('bries', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

cassini

Cassini Cylindrical Projection

Classification

Cylindrical

Identifier

`cassini`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90º from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

Parallels

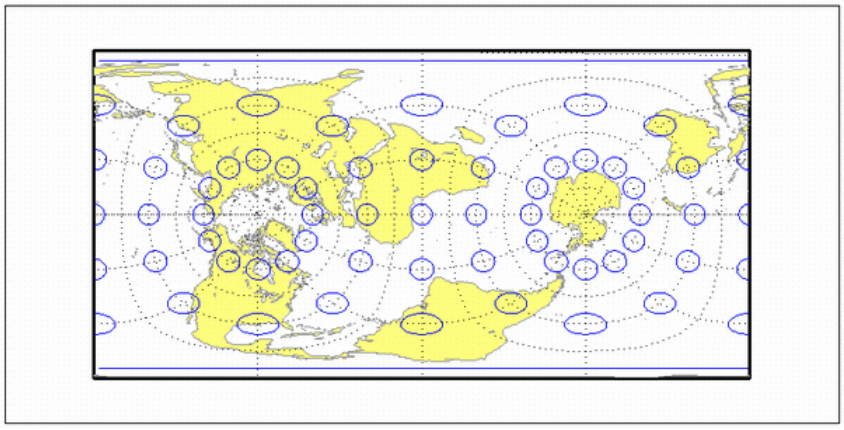
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0º.

Remarks

- This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.
- This implementation of the Cassini cylindrical projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('cassini', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See also

`cassinistd` on page 10-26

Version History

Introduced before R2006a

cassinistd

Cassini Cylindrical Projection — Standard

Identifier

`cassinistd`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90º from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

Parallels

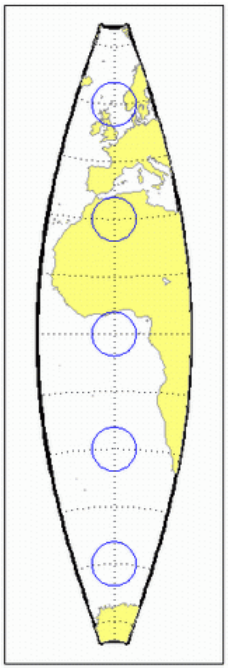
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0º.

Remarks

- This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.
- `cassinistd` implements the Cassini projection directly on a sphere or reference ellipsoid, as opposed to using the equidistant cylindrical projection in transverse mode as in function `cassini` on page 10-24. Distinct forms are used for the sphere and ellipsoid, because approximations in the ellipsoidal formulation cause it to be appropriate only within a zone that extends 3 or 4 degrees in longitude on either side of the central meridian.
- Mapping Toolbox uses a different implementation of the standard Cassini cylindrical projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('cassinistd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See also

`cassini` on page 10-24

Version History

Introduced before R2006a

ccylin

Central Cylindrical Projection

Classification

Cylindrical

Identifier

`ccylin`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, more rapidly than that of the Mercator projection.

Poles: Cannot be shown.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from the center of the Earth onto a cylinder tangent at the Equator. It is not equal-area, equidistant, or conformal. Scale is true along the Equator and constant between two parallels equidistant from the Equator. Scale becomes infinite at the poles. There is no distortion along the Equator, but it increases rapidly away from the Equator.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

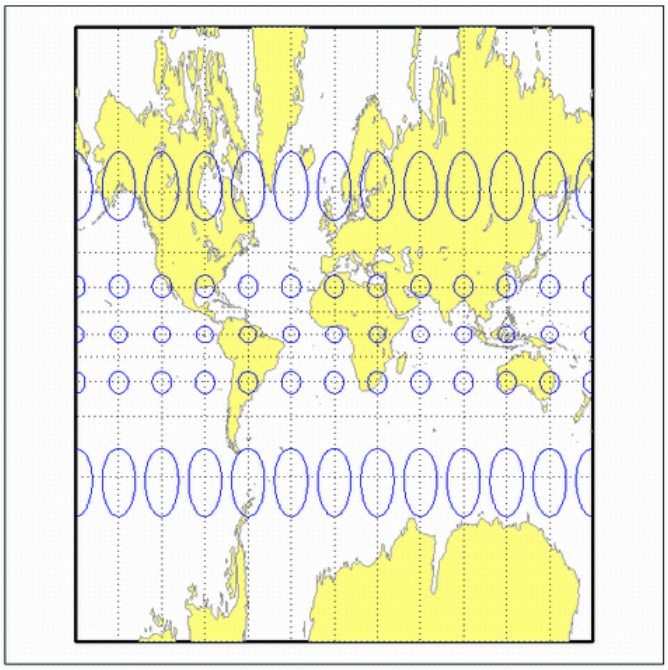
- The origin of this projection is unknown; it has little use beyond the educational aspects of its method of projection and as a comparison to the Mercator projection, which is not perspective. The transverse aspect of the central cylindrical is called the Wetch projection.
- This implementation of the central cylindrical projection is applicable only for coordinates that are referenced to a sphere.

Limitations

Data at latitudes greater than 75º is trimmed to prevent large values from dominating the display.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('ccylin', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

collig

Collignon Projection

Classification

Pseudocylindrical

Identifier

`collig`

Graticule

Meridians: Equally spaced straight lines converging at the North Pole.

Parallels: Unequally spaced straight parallel lines, farthest apart near the North Pole, closest near the South Pole

Poles: North Pole is a point, South Pole is a line 1.41 as long as the Equator.

Symmetry: About the central meridian.

Features

This is a novelty projection showing a straight-line, equal-area graticule. Scale is true along the 15º51'N parallel, constant along any parallel, and *different* for any pair of parallels. Distortion is severe in many regions, and is only absent at 15º51'N on the central meridian. This projection is not conformal or equidistant.

Parallels

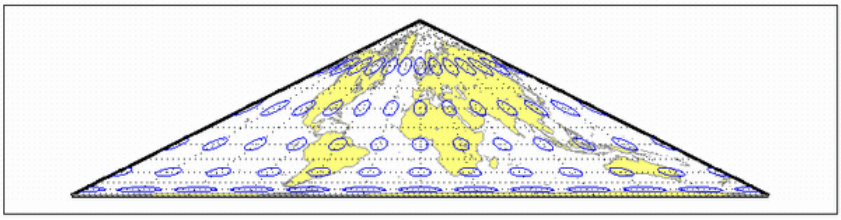
This projection has one standard parallel, which is by definition fixed at 15º51'.

Remarks

This projection was presented by Édouard Collignon in 1865.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('collig', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

craster

Craster Parabolic Projection

Classification

Pseudocylindrical

Identifier

`craster`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced parabolas intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes very gradually and is greatest near the Equator.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 36º46' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than the Sinusoidal projection. This projection is free of distortion only at the two points where the central meridian intersects the 36º46' parallels. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 36º46'.

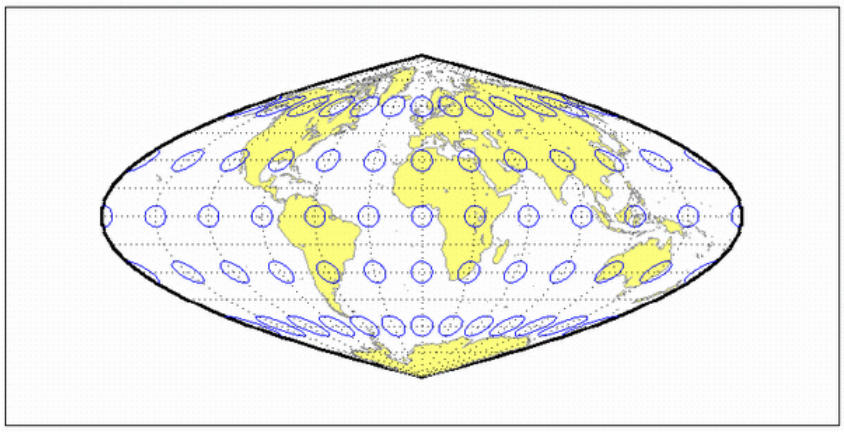
Remarks

This projection was developed by John Evelyn Edmund Craster in 1929; it was further developed by Charles H. Deetz and O.S. Adams in 1934. It was presented independently in 1934 by Putnins as his P_4 projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('craster', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eckert1

Eckert I Projection

Classification

Pseudocylindrical

Identifier

`eckert1`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced straight converging lines broken at the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 47º10' parallels and is constant along any parallel, between any pair of parallels equidistant from the Equator, and along any given meridian. It is not free of distortion at any point, and the break at the Equator introduces excessive distortion there; regardless of the appearance here, the Tissot indicatrices are of indeterminate shape along the Equator. This novelty projection is not equal-area or conformal.

Parallels

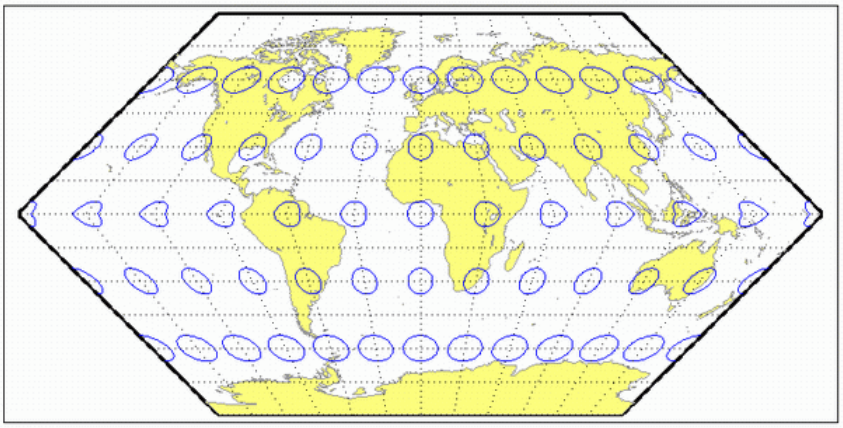
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 47º10'.

Remarks

- This projection was presented by Max Eckert in 1906.
- This implementation of the Eckert I projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Eckert I projection using the ESRI authority code 54015. For example: `projcrs(54015,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eckert2

Eckert ll Projection

Classification

Pseudocylindrical

Identifier

`eckert2`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced straight converging lines broken at the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 55º10' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is not free of distortion at any point except at 55º10'N and S along the central meridian; the break at the Equator introduces excessive distortion there. Regardless of the appearance here, the Tissot indicatrices are of indeterminate shape along the Equator. This novelty projection is not conformal or equidistant.

Parallels

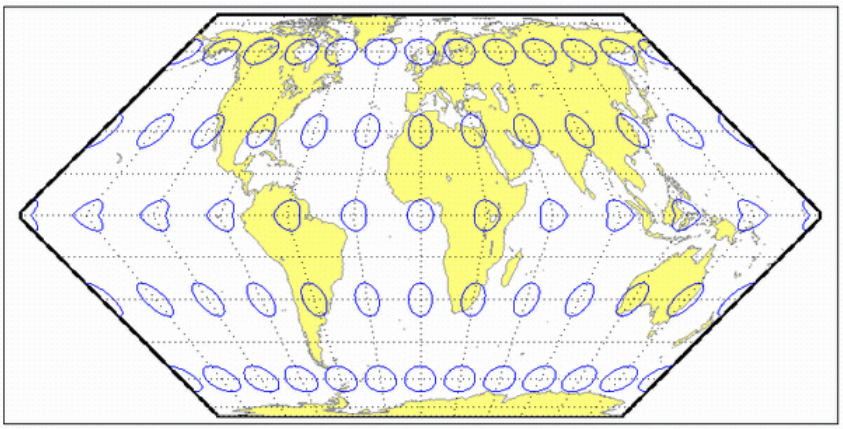
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 55º10'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert2', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eckert3

Eckert lll Projection

Classification

Pseudocylindrical

Identifier

`eckert3`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180º east and west of the central meridian, are semicircles.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 35º58' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. No point is free of all scale distortion, but the Equator is free of angular distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

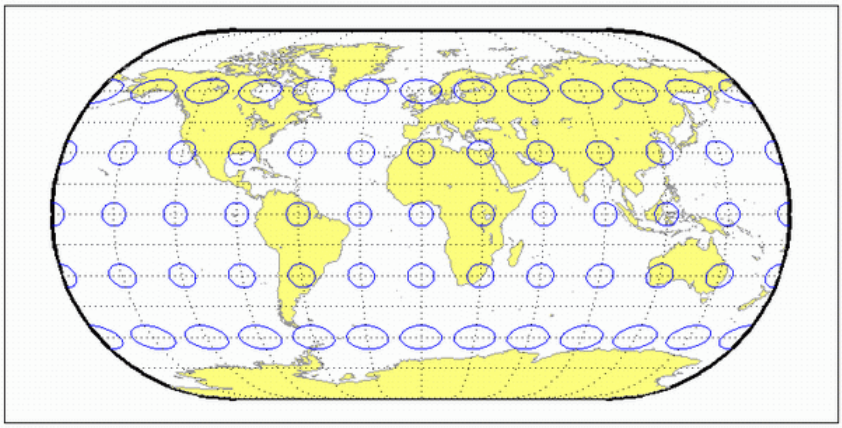
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 35º58'.

Remarks

- This projection was presented by Max Eckert in 1906.
- This implementation of the Eckert III projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Eckert III projection using the ESRI authority code 54013. For example: `projcrs(54013,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert3', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eckert4

Eckert IV Projection

Classification

Pseudocylindrical

Identifier

`eckert4`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180º east and west of the central meridian, are semicircles.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 40º30' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 40º30' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

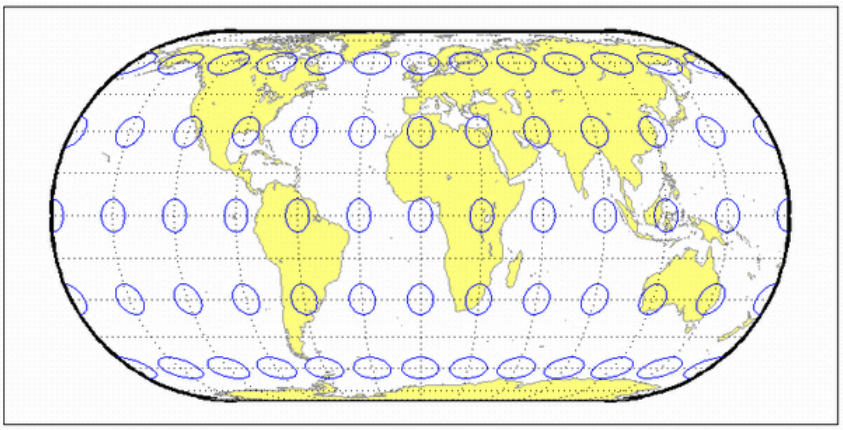
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 40º30'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert4', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eckert5

Eckert V Projection

Classification

Pseudocylindrical

Identifier

`eckert5`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is an arithmetic average of the *x* and *y* coordinates of the Sinusoidal and Plate Carrée projections. Scale is true along latitudes 37º55'N and S, and is constant along any parallel and between any pair of parallels equidistant from the Equator. There is no point free of all distortion, but the Equator is free of angular distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

This projection has one standard parallel, which is by definition fixed at 0º.

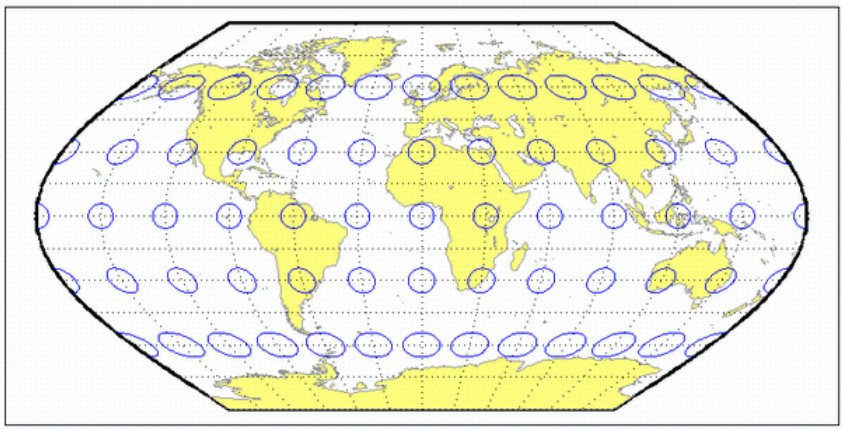
Remarks

- This projection was presented by Max Eckert in 1906.
- This implementation of the Eckert V projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Eckert V projection using the ESRI authority code 54011. For example: `projcrs(54011,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert5', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eckert6

Eckert VI Projection

Classification

Pseudocylindrical

Identifier

`eckert6`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 49º16' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 49º16' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

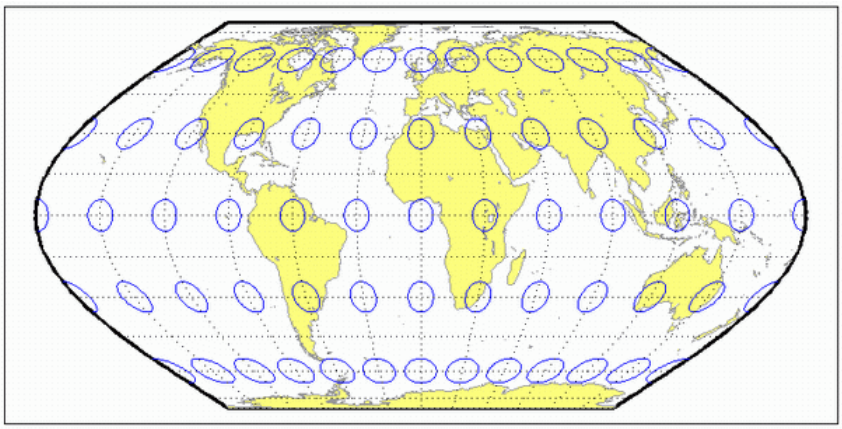
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 49º16'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert6', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eqacylin

Equal-Area Cylindrical Projection

Classification

Cylindrical

Identifier

`eqacylin`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the standard parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

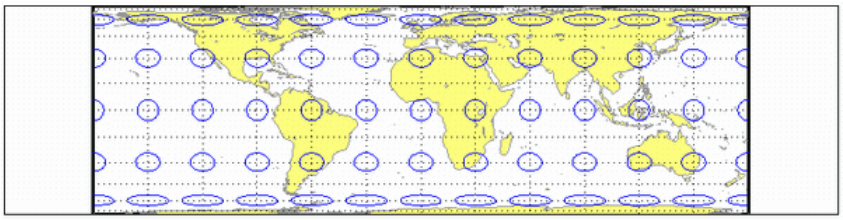
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude may be chosen; the default is arbitrarily set to 0º (the Lambert variation).

Remarks

- This projection was proposed by Johann Heinrich Lambert (1772), a prolific cartographer who proposed seven different important projections. The form of this projection tangent at the Equator is often called the Lambert Equal-Area Cylindrical projection. That and other special forms of this projection are included separately in this guide, including the Gall Orthographic, the Behrmann Cylindrical, the Balthasart Cylindrical, and the Trystan Edwards Cylindrical projections.
- Mapping Toolbox uses a different implementation of the equal-area cylindrical projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqacylin', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eqdazim

Equidistant Azimuthal Projection

Classification

Azimuthal

Identifier

`eqdazim`

Graticule

The graticule described is for the polar aspect.

Meridians: Equally spaced straight lines intersecting at a central pole. The angles between them are the true angles.

Parallels: Equally spaced circles, centered on the central pole. The entire Earth may be shown.

Poles: Central pole is a point. The opposite pole is a bounding circle with a radius twice that of the Equator.

Symmetry: About any meridian.

Features

This is an equidistant projection. It is neither equal-area nor conformal. In the polar aspect, scale is true along any meridian. The projection is distortion free only at the center point. Distortion is moderate for the inner hemisphere, but it becomes extreme in the outer hemisphere.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

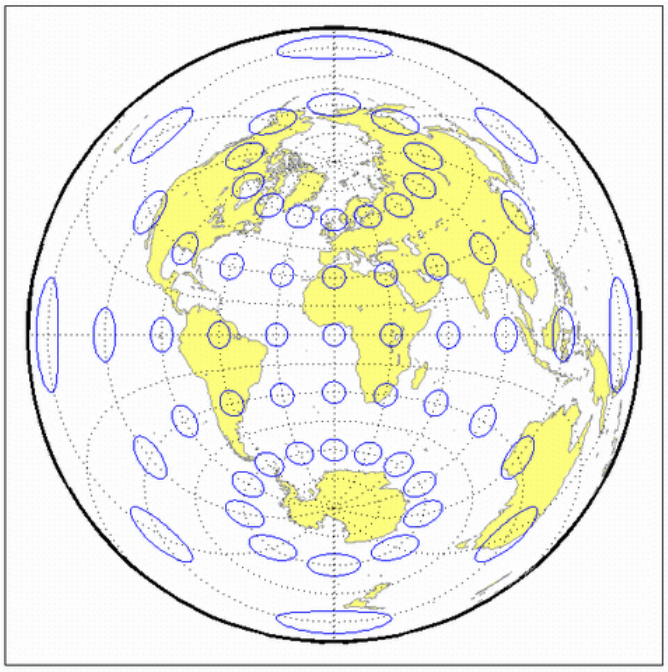
- This projection may have been first used by the ancient Egyptians for star charts. Several cartographers used it during the sixteenth century, including Guillaume Postel, who used it in 1581. Other names for this projection include Postel and Zenithal Equidistant.
- Mapping Toolbox uses a different implementation of the equidistant azimuthal projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.
- The implementation of the equidistant azimuthal projection for displaying coordinates on `axesm`-based maps is applicable only for coordinates that are referenced to a sphere. The implementation of the equidistant azimuthal projection for projecting coordinates using the `projfwd` or `projinv` function is applicable for coordinates referenced to either a sphere or an ellipsoid.

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdazim', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eqdconic

Equidistant Conic Projection

Classification

Conic

Identifier

`eqdconic`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Equally spaced concentric circular arcs centered on the point of meridional convergence.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

Scale is true along each meridian and the one or two selected standard parallels. Scale is constant along any parallel. This projection is free of distortion along the two standard parallels. Distortion is constant along any other parallel. This projection provides a compromise in distortion between conformal and equal-area conic projections, of which it is neither.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and an Equidistant Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then an Equidistant Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is so chosen, the cone becomes a cylinder and a Plate Carrée projection results. If two parallels equidistant from the Equator are chosen as the standard parallels, an Equidistant Cylindrical projection results. The default parallels are [15 75].

Remarks

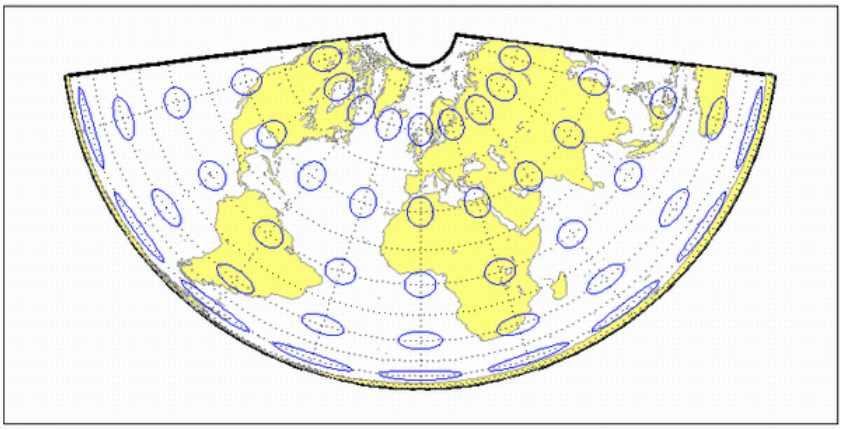
In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdconic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqdconicstd` on page 10-52

Version History

Introduced before R2006a

eqdconicstd

Equidistant Conic Projection — Standard

Identifier

`eqdconicstd`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Equally spaced concentric circular arcs centered on the point of meridional convergence.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

`eqdconicstd` implements the Equidistant Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `eqdconic` on page 10-50 for an alternative implementation based on rotating the rectifying sphere.

Scale is true along each meridian and the one or two selected standard parallels. Scale is constant along any parallel. This projection is free of distortion along the two standard parallels. Distortion is constant along any other parallel. This projection provides a compromise in distortion between conformal and equal-area conic projections, of which it is neither.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and an Equidistant Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then an Equidistant Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is so chosen, the cone becomes a cylinder and a Plate Carrée projection results. If two parallels equidistant from the Equator are chosen as the standard parallels, an Equidistant Cylindrical projection results. The default parallels are [15 75].

Remarks

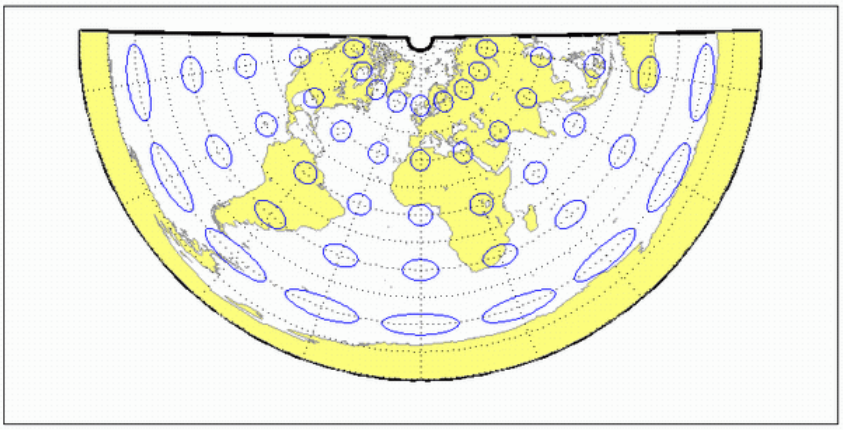
- In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.
- Mapping Toolbox uses a different implementation of the standard equidistant conic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdconicstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqdconic` on page 10-50

Version History

Introduced before R2006a

eqdcylin

Equidistant Cylindrical Projection

Classification

Cylindrical

Identifier

`eqdcylin`

Graticule

Meridians: Equally spaced straight parallel lines more than half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder secant at the standard parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the standard parallels and is constant along any parallel and along the parallel of opposite sign.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude can be chosen; the default is arbitrarily set to 30º.

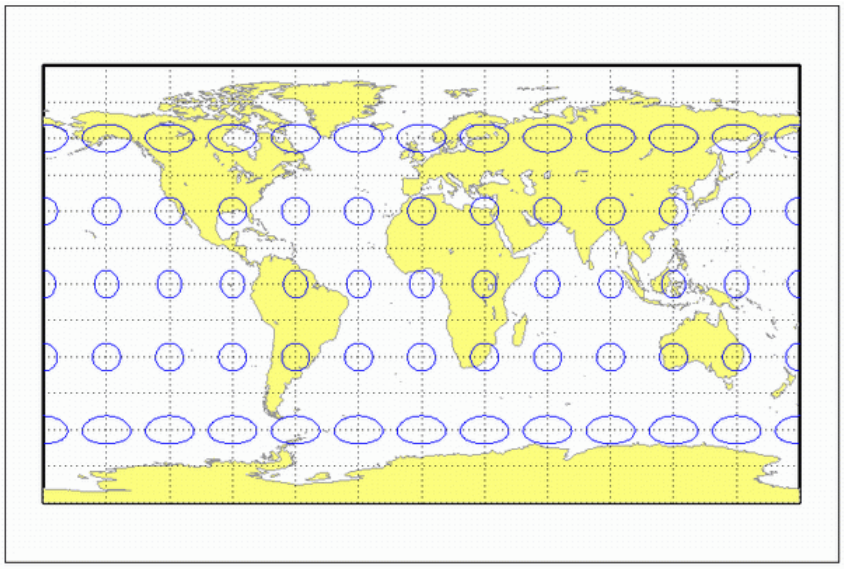
Remarks

- This projection was first used by Marinus of Tyre about A.D. 100. Special forms of this projection are the Plate Carrée, with a standard parallel at 0º, and the Gall Isographic, with standard parallels at 45ºN and S. Other names for this projection include Equirectangular, Rectangular, Projection of Marinus, *La Carte Parallélogrammatique*, and *Die Rechteckige Plattkarte*.
- By default, the standard parallels are at +/- 30 degrees in geodetic latitude.
- When projecting a sphere, the origin vector is used to specify a triaxial rigid-body rotation.
- When projecting an ellipsoid:

  - The origin longitude (2nd element of the origin vector) determines which meridian maps to the line *x* == false easting
  - The origin latitude (1st element of the origin vector) is used to shift the natural origin off the equator via a constant *y*-offset, in addition to any false northing that may be specified.
  - The grid convergence is fixed at 0, even if the 3rd element of the origin vector is nonzero.
- Mapping Toolbox uses a different implementation of the equidistant cylindrical projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdcylin', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

fournier

Fournier Projection

Classification

Pseudocylindrical

Identifier

`fournier`

Graticule

Meridians: Equally spaced elliptical curves converging at the poles.

Parallels: Straight lines.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is equal-area. Scale is constant along any parallel or pair of parallels equidistant from the Equator. This projection is neither equidistant nor conformal.

Parallels

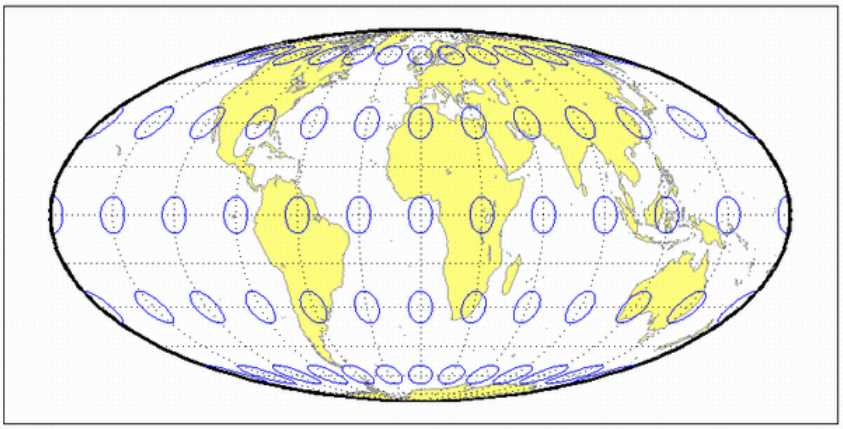
There is no standard parallel for this projection.

Remarks

This projection was first described in 1643 by Georges Fournier. This is actually his second projection, the Fournier II.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('fournier', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

giso

Gall Isographic Projection

Classification

Cylindrical

Identifier

`giso`

Graticule

Meridians: Equally spaced straight parallel lines more than half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder secant at the 45º parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the two standard parallels, and is constant along any parallel and along the parallel of opposite sign.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

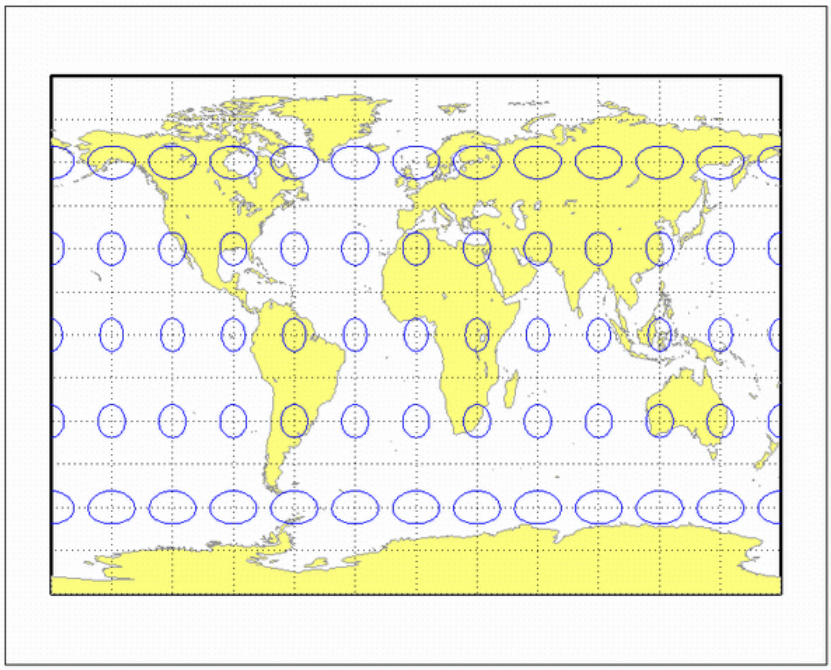
Remarks

- This projection is a specific case of the Equidistant Cylindrical projection, with standard parallels at 45ºN and S.
- On the sphere, this projection can have an arbitrary, oblique aspect, as controlled by the `Origin` property of the `axesm`-based map. On the ellipsoid, only the equatorial aspect is supported.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('giso', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

gortho

Gall Orthographic Projection

Classification

Cylindrical

Identifier

`gortho`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 45º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

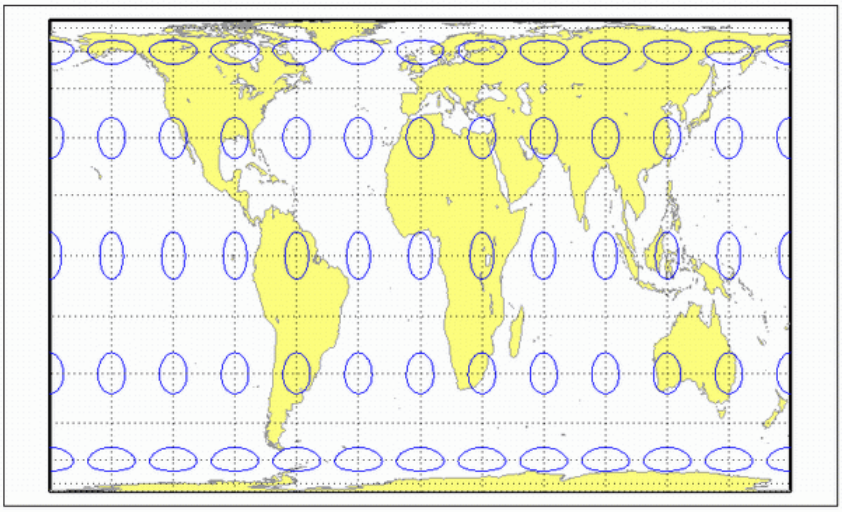
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

Remarks

This projection is named for James Gall, who originated it in 1855 and is a special form of the Equal-Area Cylindrical projection secant at 45ºN and S. This projection is also known as the Peters projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('gortho', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

gstereo

Gall Stereographic Projection

Classification

Cylindrical

Identifier

`gstereo`

Graticule

Meridians: Equally spaced straight parallel lines 0.77 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 45º parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

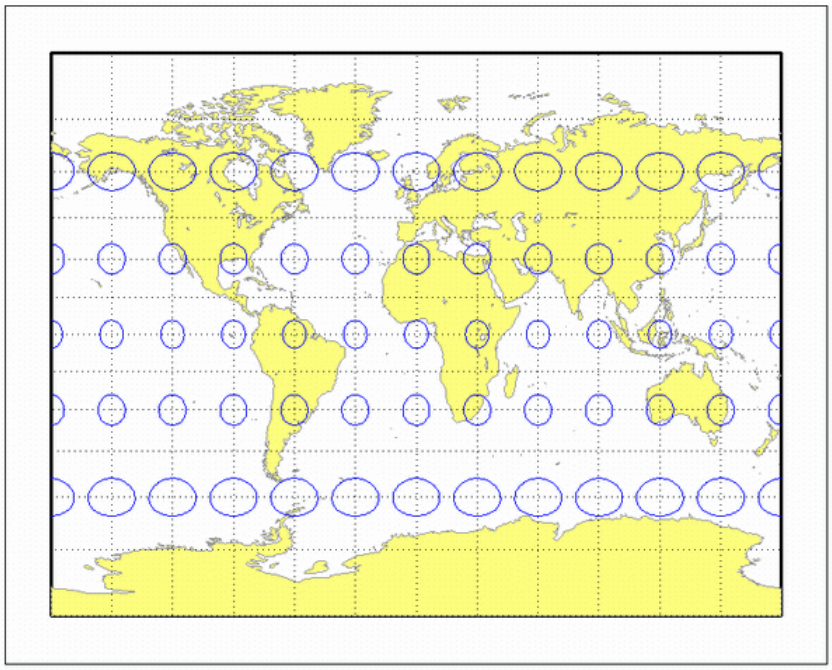
Remarks

- This projection was presented by James Gall in 1855. It is also known simply as the Gall projection. It is a special form of the Braun Perspective Cylindrical projection secant at 45ºN and S.
- This implementation of the Gall stereographic projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Gall stereographic projection using the ESRI authority code 54016. For example: `projcrs(54016,'Authority','ESRI')`.

- Mapping Toolbox uses a different implementation of the Gall stereographic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('gstereo', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

globe

Frame for 3-D map display on `axesm`-based map

Classification

Spherical

Identifier

`globe`

Graticule

This map display is based on a coordinate transformation and is not a true map projection. Meridians, parallels, and displayed map data appear in a 3-D view that depends on the view and camera settings of the `axesm`-based map. Change the view interactively or by using the `view` function. Change the camera settings using the `camposm`, `camtargm`, and `camupm` functions.

Features

In the 3-D sense, `globe` is true in scale, equal-area, conformal, minimum error, and equidistant everywhere.

While `globe` has none of the distortions inherent in planar projections, it cannot be displayed without distortion or in its entirety. In order to render the globe in a figure window, either a perspective or orthographic transformation must be applied, both of which necessarily involve setting a viewpoint, hiding the back side and distortions of shape, scale, and angles.

Parallels

The globe requires no standard parallels.

Remarks

The `globe` display allows you to visualize terrain relief or other data for an entire planet viewed from space. Its underlying transformation maps latitude, longitude, and elevation to a 3-D Cartesian frame. The `globe` display is different from other transformations because it can render relative relief of elevations above, below, or on a sphere.

When displayed, the globe looks like an orthographic azimuthal projection, provided that the `Projection` property of the `axesm`-based map is set to `'orthographic'`.

Examples

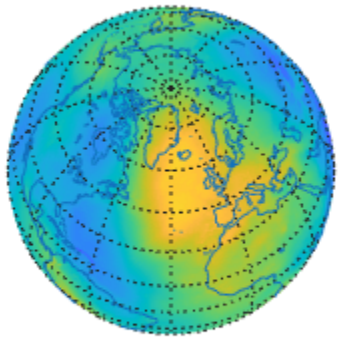
Display Geoid Heights on Globe

Display geoid heights from the EGM96 geoid model over a 3-D globe. First, get geoid heights and a geographic postings reference object. Load coastline latitude and longitude data.

```
[N,R] = egm96geoid;
load coastlines
```

Create a frame for the 3-D globe display using `axesm`. Set the line of sight for the globe using `view`. Turn off the axes background using `axis off`. Then, display the geoid heights and coastline data.

```
axesm('globe','Grid','on')
view(60,60)
axis off
meshm(N,R)
plotm(coastlat,coastlon)
```



Display Polygon on Globe

Display a polygon on a globe by converting the polygon to a data grid.

Create a sample polygon that contains a hole and rests on the surface of the globe. To do this, generate the vertices of its external and internal boundaries using the `outlinegeoquad` function. Specify the geographic limits as the first two arguments, and the vertex spacing in degrees as the next two arguments. Reverse the order of the internal boundary vertices using the `flip` function, so they are in a counterclockwise order.

```
[latE,lonE] = outlinegeoquad([-35 35],[-30 30],0.25,0.25);
[latI,lonI] = outlinegeoquad([-15 15],[-15 15],0.25,0.25);
latI = flip(latI);
lonI = flip(lonI);
```

Combine the vertices into a single list by separating the boundaries with `NaN` values.

```
lat = [latE NaN latI];
lon = [lonE NaN lonI];
```

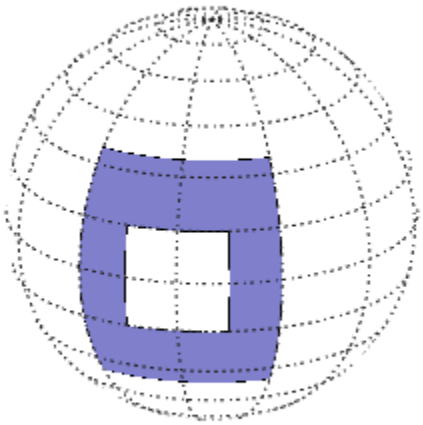
The vectors `lat` and `lon` represent the boundaries of a polygon that contain a hole. Display the boundaries on the globe as a filled polygon by converting the polygon to a data grid.

To do this, create a geographic cells reference object for the globe and a grid of ones. Replace elements of the grid with the polygon data using the `vec2mtx` function. The new grid contains 0s to indicate the inside region of the polygon, 1s to indicate the boundaries, and 2s to indicate the outside region of the polygon.

```
R = georefcells([-90 90],[-180 180],0.25,0.25);
V = ones(R.RasterSize);
[V,R] = vec2mtx(lat,lon,V,R,'filled');
```

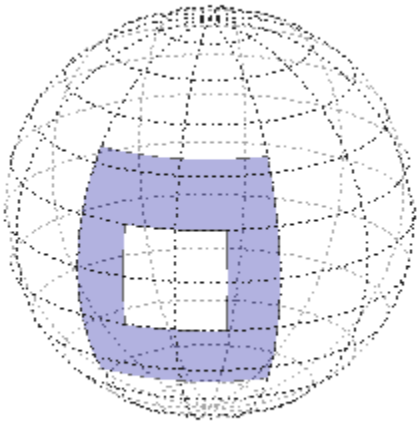
Create a globe using the `axesm` function. Display the data grid as an image using the `geoshow` function. Adjust the colormap so the inside region of the polygon is purple and the outside region is white. Change the camera line of sight using the `view` function, so the polygon is displayed on the near side of the globe.

```
axesm('globe','Grid','on')
geoshow(V,R,'DisplayType','texturemap')
colormap([0.5 0.5 0.8; 0 0 0; 1 1 1])
axis off
view(100,20)
```

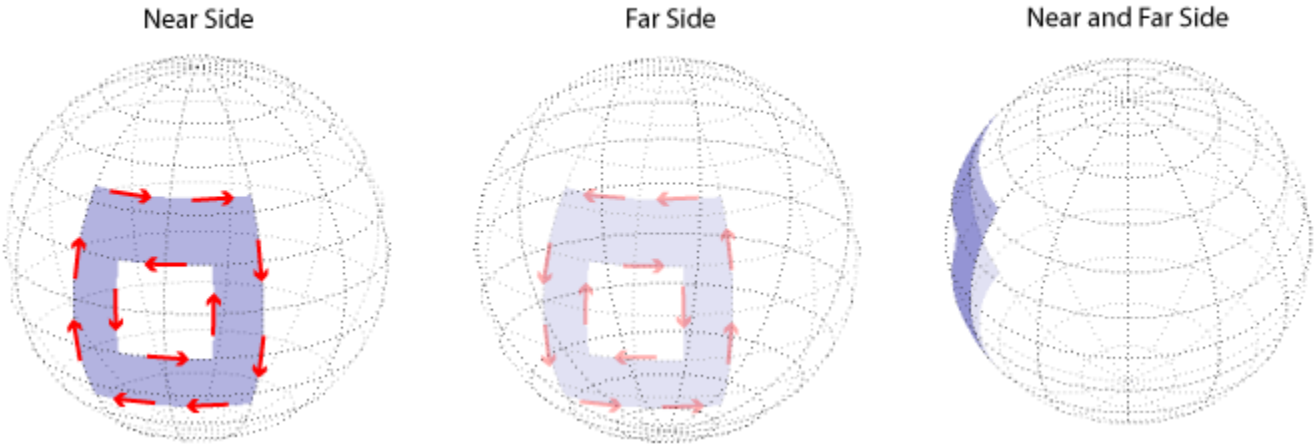


The appearance of polygons on the globe is dependent on the camera line of sight and the globe transparency. For example, make the globe slightly transparent using the `alpha` function.

```
alpha(0.6)
```



When you view the polygon from the near side of the globe, the external boundary vertices appear in a clockwise order. When you view the polygon from the far side of the globe, the external boundary vertices appear in a counterclockwise order. When you rotate the globe so the polygon appears on both the near side and far side, then the polygon appears to intersect itself.



Version History

Introduced before R2006a

See Also

`geoshow` | `outlinegeoquad` | `axesm` | `vec2mtx`

Topics
“Create and Display Polygons” on page 6-17

gnomonic

Gnomonic Projection

Classification

Azimuthal

Identifier

`gnomonic`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing increases rapidly away from this pole. The Equator and the opposite hemisphere cannot be shown

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection from the center of the globe on a plane tangent at the center point, which is a pole in the common polar aspect, but can be any point. Less than one hemisphere can be shown with this projection, regardless of its center point. The significant property of this projection is that all great circles are straight lines. This is useful in navigation, as a great circle is the shortest path between two points on the globe. Only the center point enjoys true scale and zero distortion. This projection is neither conformal nor equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

- This projection may have been first developed by Thales around 580 B.C. Its name is derived from the gnomon, the face of a sundial, since the meridians radiate like hour markings. This projection is also known as a gnomic or central projection.
- This implementation of the gnomonic projection is applicable only for coordinates that are referenced to a sphere.

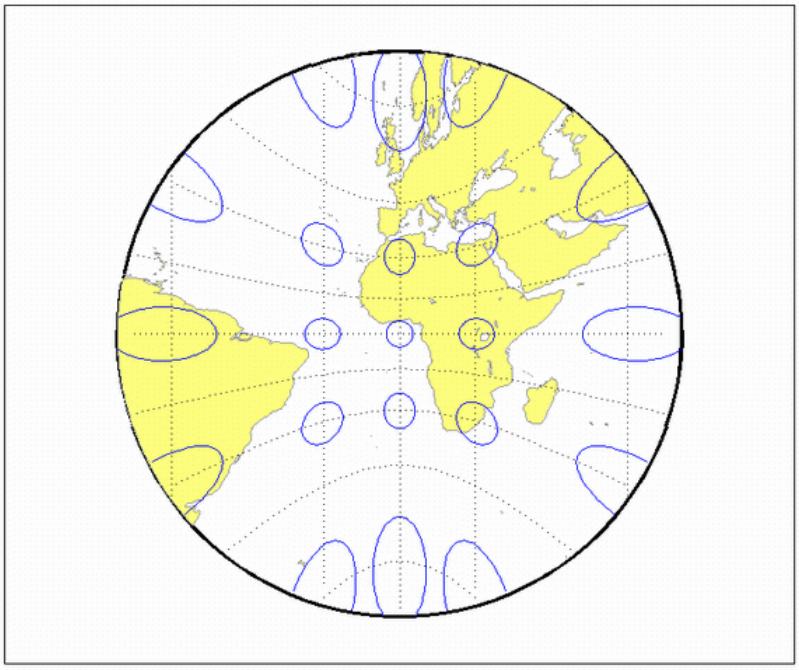
- Mapping Toolbox uses a different implementation of the gnomonic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data greater than 65º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('gnomonic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

goode

Goode Homolosine Projection

Classification

Pseudocylindrical

Identifier

goode

Graticule

Central Meridian: Straight line 0.44 as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves between the 40º44'11.8'' parallels and elliptical arcs elsewhere, all concave toward the central meridian. The result is a slight, visible bend in the meridians at 40º44'11.8'' N and S.

Parallels: Straight parallel lines, perpendicular to the central meridian. Equally spaced between the 40º44'11.8'' parallels, with gradually decreasing spacing outside these parallels.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along all parallels and the central meridian between 40º44'11.8'' N and S, and is constant along any parallel and between any pair of parallels equidistant from the Equator for all latitudes. Its distortion is identical to that of the Sinusoidal projection between 40º44'11.8'' N and S, and to that of the Mollweide projection elsewhere. This projection is not conformal or equidistant.

Parallels

This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

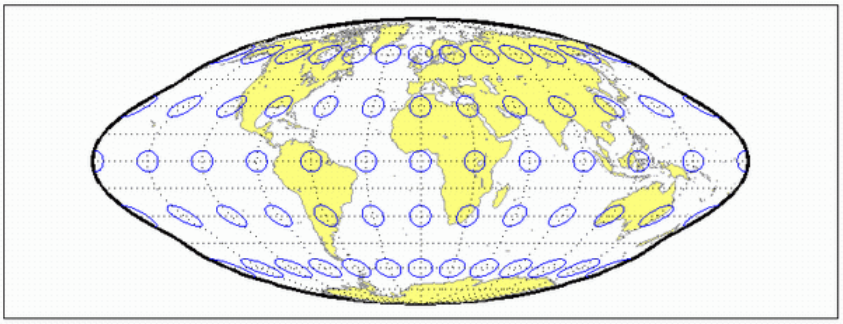
This projection was developed by J. Paul Goode in 1916. It is sometimes called simply the Homolosine projection, and it is usually used in an interrupted form. It is a merging of the Sinusoidal and Mollweide projections.

Limitations

This projection is available in an uninterrupted form only.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('goode', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

hammer

Hammer Projection

Classification

Modified Azimuthal

Identifier

`hammer`

Graticule

Meridians: Central meridian is a straight line half the length of the Equator. Other meridians are complex curves, equally spaced along the Equator, and concave toward the central meridian.

Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is equal-area. The only point free of distortion is the center point. Distortion of shape is moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

Parallels

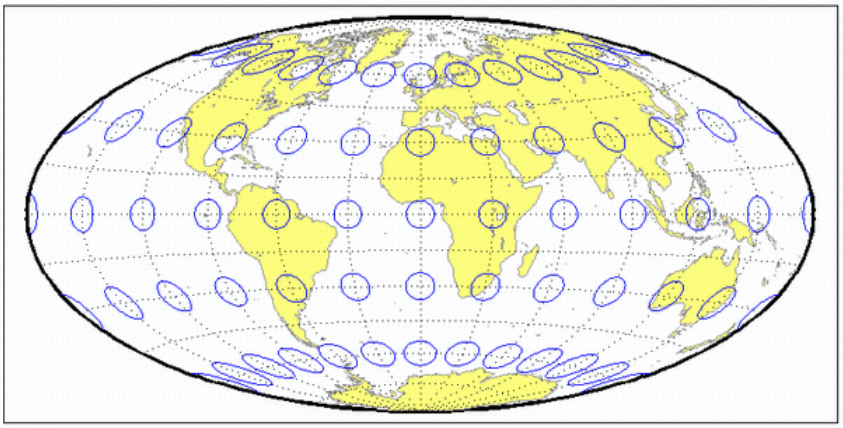
There is no standard parallel for this projection.

Remarks

- This projection was presented by H. H. Ernst von Hammer in 1892. It is a modification of the Lambert Azimuthal Equal Area projection. Inspired by Aitoff projection, it is also known as the Hammer-Aitoff. It in turn inspired the Briesemeister, a modified oblique Hammer projection. John Bartholomew's Nordic projection is an oblique Hammer centered on 45 degrees north and the Greenwich meridian. The Hammer projection is used in whole-world maps and astronomical maps in galactic coordinates.
- This implementation of the Hammer projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('hammer', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

hatano

Hatano Asymmetrical Equal-Area Projection

Classification

Pseudocylindrical

Identifier

`hatano`

Graticule

Central Meridian: Straight line 0.48 as long as the Equator.

Other Meridians: Equally spaced elliptical arcs concave toward the central meridian. The eccentricity of each ellipse changes at the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is not symmetrical about the Equator.

Poles: The North Pole is a line two-thirds the length of the Equator; the South Pole is a line three-fourths the length of the Equator.

Symmetry: About the central meridian but *not* the Equator.

Features

This is an equal-area projection. Scale is true along 40º42'N and 38º27'S, and is constant along any parallel but generally *not* between pairs of parallels equidistant from the Equator. It is free of distortion only along the central meridian at 40º42'N and 38º27'S. This projection is not conformal or equidistant.

Parallels

Because of the asymmetrical nature of this projection, two standard parallels must be specified. The standard parallels are by definition fixed at 40º42'N and 38º27'S.

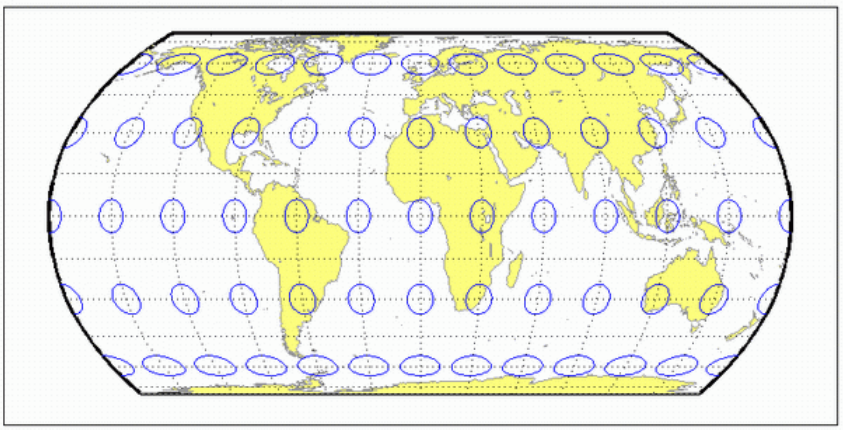
Remarks

This projection was presented by Masataka Hatano in 1972.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('hatano', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

kavrsky5

Kavraisky V Projection

Classification

Pseudocylindrical

Identifier

`kavrsky5`

Graticule

Meridians: Complex curves converging at the poles. A sine function is used for *y*, but the meridians are not sine curves.

Parallels: Unequally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and the central meridian.

Features

This is an equal-area projection. Scale is true along the fixed standard parallels at 35º, and 0.9 true along the Equator. This projection is neither conformal nor equidistant.

Parallels

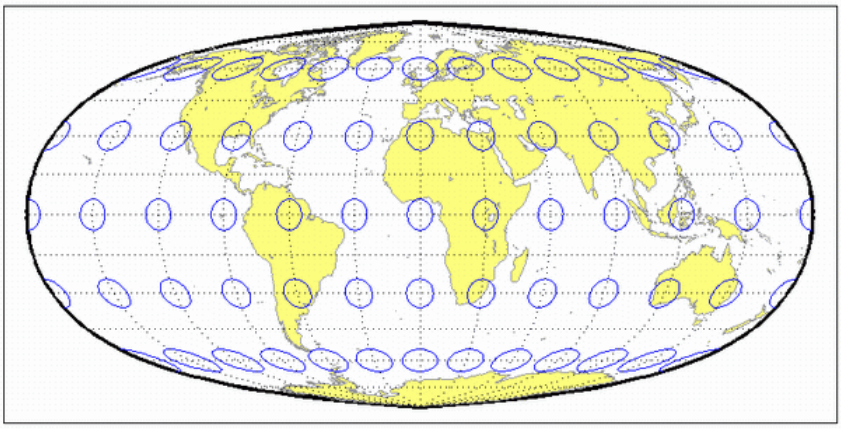
The fixed standard parallels are at 35º.

Remarks

This projection was described by V. V. Kavraisky in 1933.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('kavrsky5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

kavrsky6

Kavraisky VI Projection

Classification

Pseudocylindrical

Identifier

`kavrsky6`

Graticule

Central Meridian: Straight line half the length of the Equator.

Meridians: Sine curves (60º segments).

Parallels: Unequally spaced straight lines.

Poles: Straight lines half the length of the Equator.

Symmetry: About the Equator and the central meridian.

Features

This is an equal-area projection. Scale is constant along any parallel or pair of equidistant parallels. This projection is neither conformal nor equidistant.

Parallels

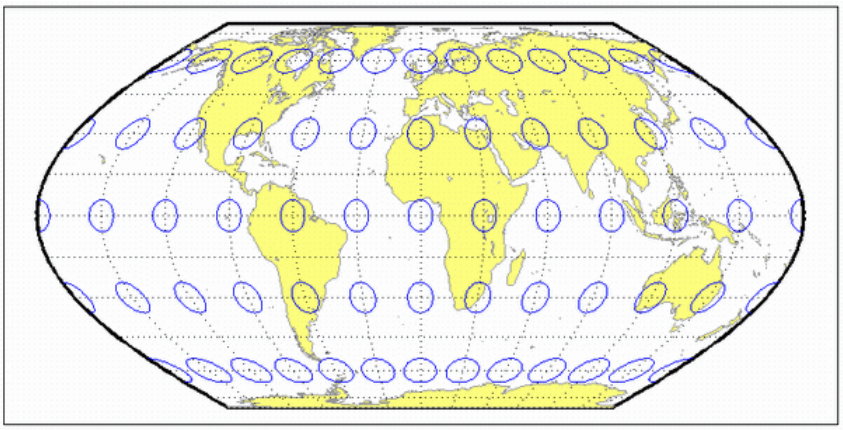
There are no standard parallels for this projection.

Remarks

This projection was described by V. V. Kavraisky in 1936. It is also called the Wagner I, for Karlheinz Wagner, who described it in 1932.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('kavrsky6', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

eqaazim

Lambert Azimuthal Equal-Area Projection

Classification

Azimuthal

Identifier

`eqaazim`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. The entire Earth can be shown. Spacing decreases away from the central pole.

Pole: The central pole is a point; the other pole is a bounding circle with 1.41 the radius of the Equator.

Symmetry: About any meridian.

Features

This nonperspective projection is equal-area. Only the center point is free of distortion, but distortion is moderate within 90º of this point. Scale is true only at the center point, increasing tangentially and decreasing radially with distance from the center point. This projection is neither conformal nor equidistant.

Parallels

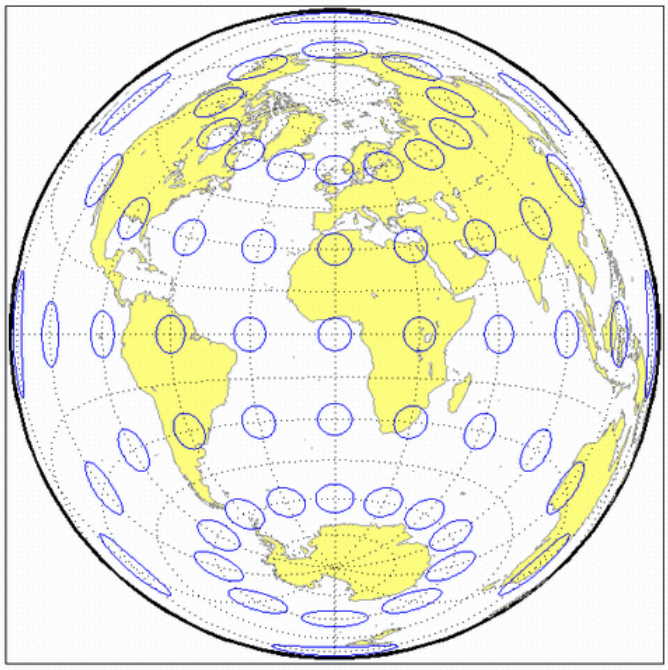
There are no standard parallels for azimuthal projections.

Remarks

- This projection was presented by Johann Heinrich Lambert in 1772. It is also known as the Zenithal Equal-Area and the Zenithal Equivalent projection, and the Lorgna projection in its polar aspect.
- Mapping Toolbox uses a different implementation of the Lambert azimuthal equal-area projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqaazim', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



lambert

Lambert Conformal Conic Projection

Classification

Conic

Identifier

`lambert`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.

Poles: The pole nearest a standard parallel is a point, the other cannot be shown.

Symmetry: About any meridian.

Features

Scale is true along the one or two selected standard parallels. Scale is constant along any parallel and is the same in every direction at any point. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is conformal everywhere but the poles; it is neither equal-area nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Stereographic Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Conformal Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator or two parallels equidistant from the Equator are chosen as the standard parallels, the cone becomes a cylinder, and a Mercator projection results. The default parallels are [15 75].

Remarks

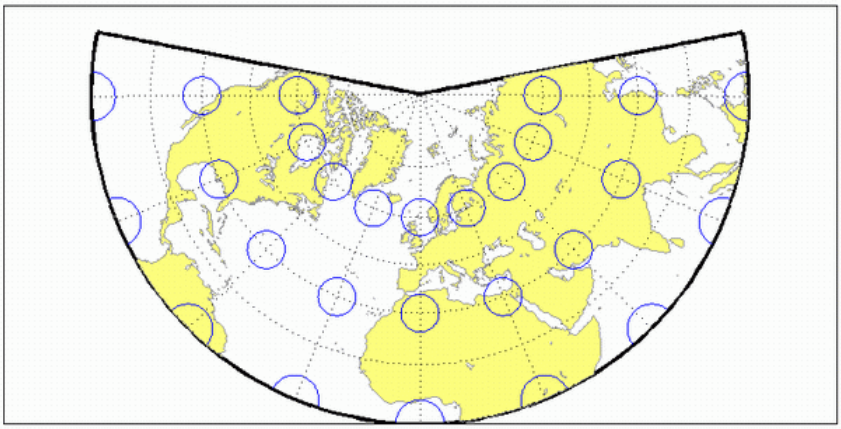
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed. The default map limits are [0 90] to avoid extreme area distortion.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('lambert', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`lambertstd` on page 10-85

lambertstd

Lambert Conformal Conic Projection — Standard

Classification

Conic

Identifier

`lambertstd`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.

Poles: The pole nearest a standard parallel is a point, the other cannot be shown.

Symmetry: About any meridian.

Features

`lambertstd` implements the Lambert Conformal Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `lambert` on page 10-83 for an alternative implementation based on rotating the authalic sphere.

Scale is true along the one or two selected standard parallels. Scale is constant along any parallel and is the same in every direction at any point. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is conformal everywhere but the poles; it is neither equal-area nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Stereographic Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Conformal Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator or two parallels equidistant from the Equator are chosen as the standard parallels, the cone becomes a cylinder, and a Mercator projection results. The default parallels are [15 75].

Remarks

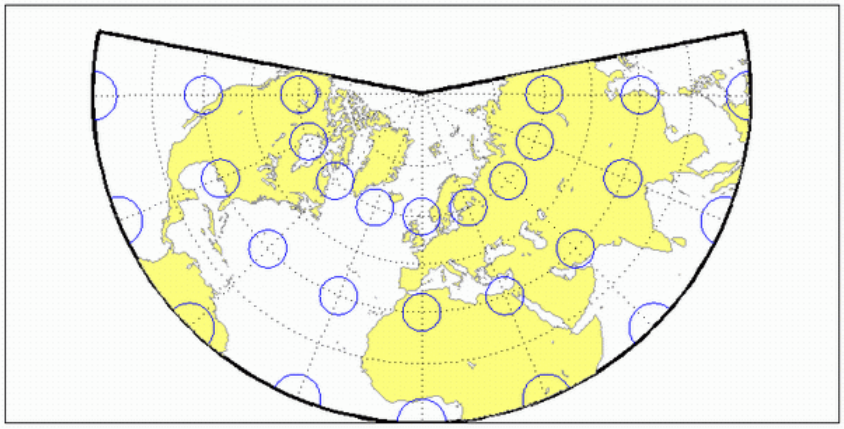
- This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.
- Mapping Toolbox uses a different implementation of the standard Lambert conformal conic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed. The default map limits are [0 90] to avoid extreme area distortion.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('lambertstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`lambert` on page 10-83

Version History

Introduced before R2006a

lambcyln

Lambert Equal-Area Cylindrical Projection

Classification

Cylindrical

Identifier

`lambcyln`

Graticule

Meridians: Equally spaced straight parallel lines 0.32 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder tangent at the Equator. It is equal-area, but distortion of shape increases with distance from the Equator. Scale is true along the Equator and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

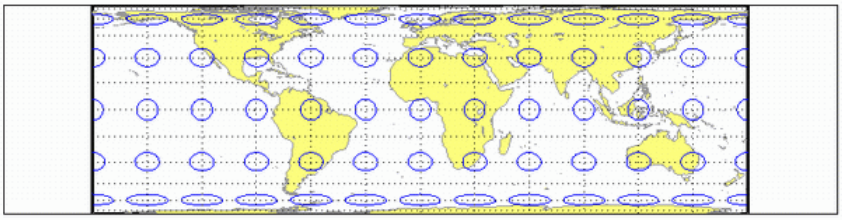
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

This projection is named for Johann Heinrich Lambert and is a special form of the Equal-Area Cylindrical projection tangent at the Equator.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('lambcyln', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

loximuth

Loximuthal Projection

Classification

Pseudocylindrical

Identifier

`loximuth`

Graticule

Central Meridian: Straight line at least half as long as the Equator. Actual length depends on the choice of central latitude. Length is 0.5 when the central latitude is the Equator, for example, and 0.65 for central latitudes of 40º.

Other Meridians: Complex curves intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian. Symmetry about the Equator only when it is the central latitude.

Features

This projection has the special property that from the central point (the intersection of the central latitude with the central meridian), rhumb lines (loxodromes) are shown as straight, true to scale, and correct in azimuth from the center. This differs from the Mercator projection, in that rhumb lines are here shown in true scale and that unlike the Mercator, this projection does not maintain true azimuth for all points along the rhumb lines. Scale is true along the central meridian and is constant along any parallel, but not, generally, between parallels. It is free of distortion only at the central point and can be severely distorted in places. However, this projection is designed for its specific special property, in which distortion is not a concern.

Parallels

For this projection, only one standard parallel is specified: the central latitude described above. Specification of this central latitude defines the center of the loximuthal projection. The default value is 0º.

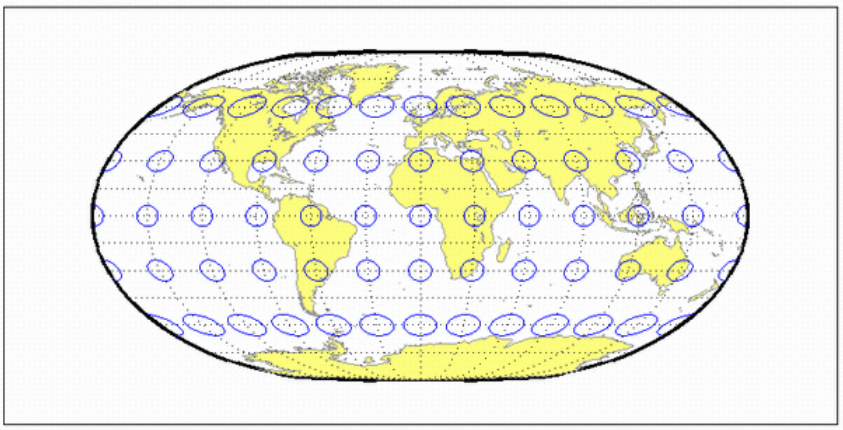
Remarks

- This projection was presented by Karl Siemon in 1935 and independently by Waldo R. Tobler in 1966. The Bordone Oval projection of 1520 was very similar to the Equator-centered loximuthal.

- This implementation of the loximuthal projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the loximuthal projection using the ESRI authority code 54023. For example: `projcrs(54023,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('loximuth', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

flatplrp

McBryde-Thomas Flat-Polar Parabolic Projection

Classification

Pseudocylindrical

Identifier

`flatplrp`

Graticule

Central Meridian: Straight line 0.48 as long as the Equator.

Other Meridians: Equally spaced parabolic curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 45º30' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the pointed-polar projections. It is free of distortion only at the two points where the central meridian intersects the 45º30' parallels. This projection is not conformal or equidistant.

Parallels

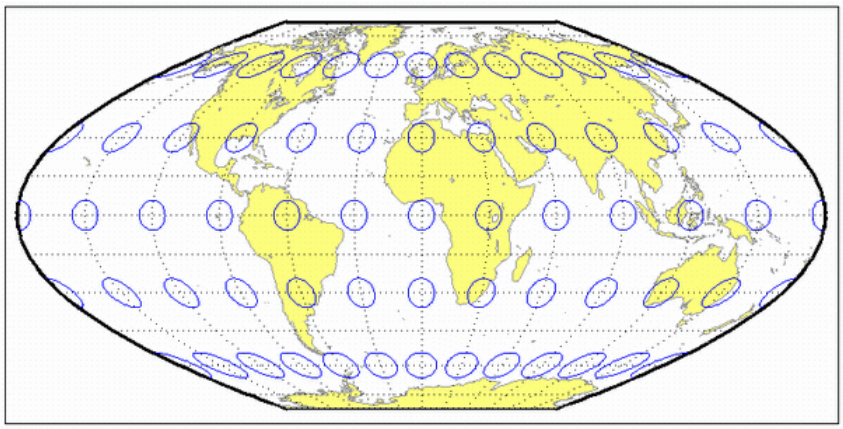
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 45º30'.

Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrp', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

flatplrq

McBryde-Thomas Flat-Polar Quartic Projection

Classification

Pseudocylindrical

Identifier

`flatplrq`

Graticule

Central Meridian: Straight line 0.45 as long as the Equator.

Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 33º45' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the pointed-polar projections. It is free of distortion only at the two points where the central meridian intersects the 33º45' parallels. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 33º45'.

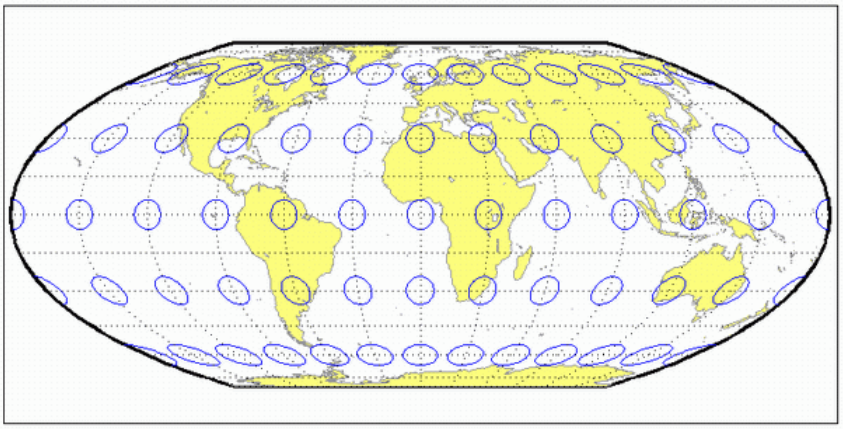
Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949, and is also known simply as the Flat-Polar Quartic projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrq', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

flatplrs

McBryde-Thomas Flat-Polar Sinusoidal Projection

Classification

Pseudocylindrical

Identifier

`flatplrs`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is equal-area. Scale is true along the 55º51' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the central meridian intersects the 55º51' parallels. This projection is not conformal or equidistant.

Parallels

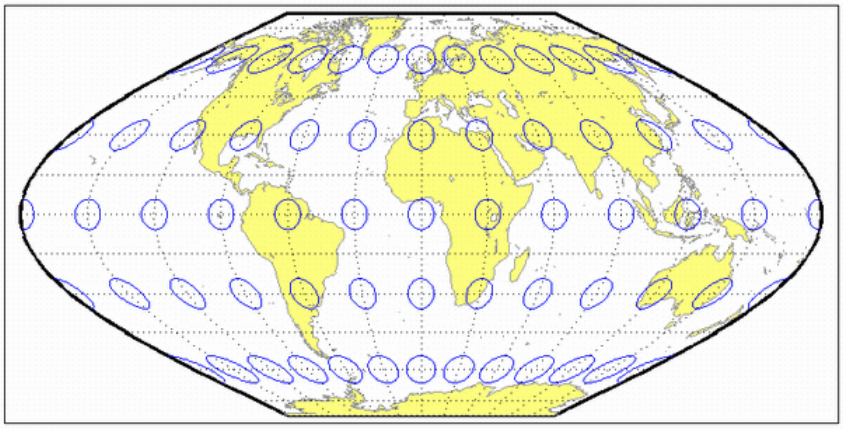
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 55º51'.

Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrs', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

mercator

Mercator Projection

Classification

Cylindrical

Identifier

`mercator`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Cannot be shown.

Symmetry: About any meridian or the Equator.

Features

This is a projection with parallel spacing calculated to maintain conformality. It is not equal-area, equidistant, or perspective. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. It is also constant in all directions near any given point. Scale becomes infinite at the poles. The appearance of the Mercator projection is unaffected by the selection of standard parallels; they serve only to define the latitude of true scale.

The Mercator, which may be the most famous of all projections, has the special feature that all rhumb lines, or loxodromes (lines that make equal angles with all meridians, i.e., lines of constant heading), are straight lines. This makes it an excellent projection for navigational purposes. However, the extreme area distortion makes it unsuitable for general maps of large areas.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude less than 86º may be chosen; the default is arbitrarily set to 0º.

Remarks

- The Mercator projection is named for Gerardus Mercator, who presented it *for navigation* in 1569. It is now known to have been used for the Tunhuang star chart as early as 940 by Ch'ien Lo-Chih. It was first used in Europe by Erhard Etzlaub in 1511. It is also, but rarely, called the Wright projection, after Edward Wright, who developed the mathematics behind the projection in 1599.

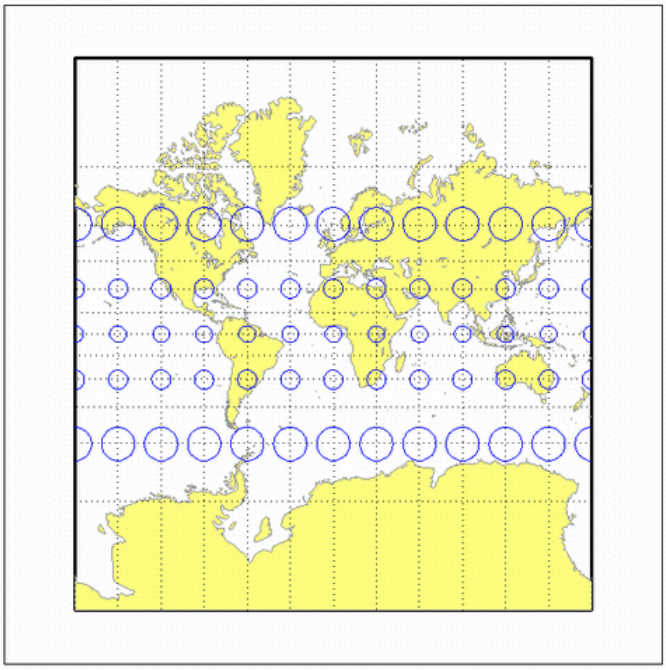
- Mapping Toolbox uses a different implementation of the Mercator projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data at latitudes greater than 86º is trimmed to prevent large *y*-values from dominating the display.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('mercator', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

miller

Miller Cylindrical Projection

Classification

Cylindrical

Identifier

`miller`

Graticule

Meridians: Equally spaced straight parallel lines 0.73 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, less rapidly than that of the Mercator projection.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection with parallel spacing calculated to maintain a look similar to the Mercator projection while reducing the distortion near the poles and allowing the poles to be displayed. It is not equal-area, equidistant, conformal, or perspective. Scale is true along the Equator and constant between two parallels equidistant from the Equator. There is no distortion near the Equator, and it increases moderately away from the Equator, but it becomes severe at the poles.

The Miller Cylindrical projection is derived from the Mercator projection; parallels are spaced from the Equator by calculating the distance on the Mercator for a parallel at 80% of the true latitude and dividing the result by 0.8. The result is that the two projections are almost identical near the Equator.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

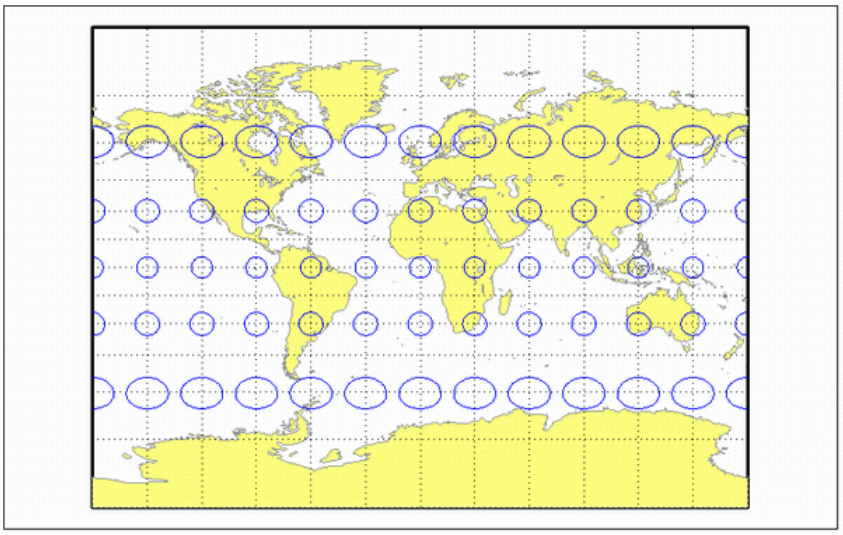
Remarks

- This projection was presented by Osborn Maitland Miller of the American Geographical Society in 1942. It is often used in place of the Mercator projection for atlas maps of the world, for which it is somewhat more appropriate.

- Mapping Toolbox uses a different implementation of the Miller projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.
- The implementation of the Miller projection for displaying coordinates on `axesm`-based maps is applicable only for coordinates that are referenced to a sphere. The implementation of the Miller projection for projecting coordinates using the `projfwd` or `projinv` function is applicable for coordinates referenced to either a sphere or an ellipsoid.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('miller', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

mollweid

Mollweide Projection

Classification

Pseudocylindrical

Identifier

`mollweid`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Meridians 90º east and west of the central meridian form a circle. The others are equally spaced semiellipses intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator, but the spacing changes gradually.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 40º44' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 40º44' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 40º44'.

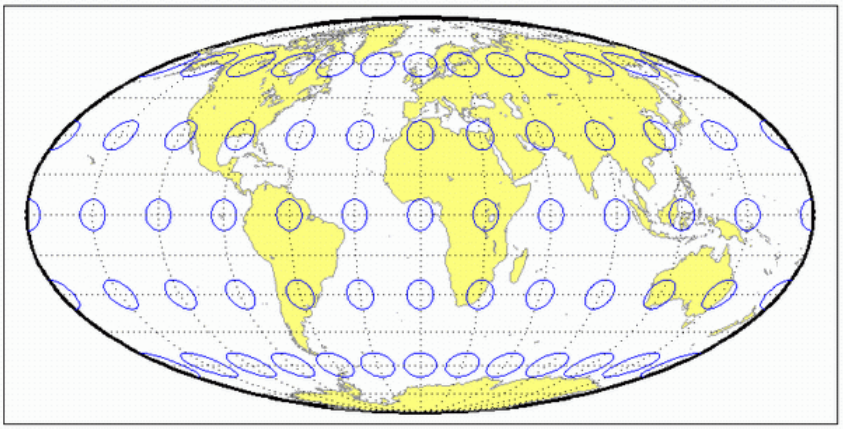
Remarks

This projection was presented by Carl B. Mollweide in 1805. Its other names include the Homolographic, the Homalographic, the Babinet, and the Elliptical projections. It is occasionally used for thematic world maps, and it is combined with the Sinusoidal to produce the Goode Homolosine projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('mollweid', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

murdoch1

Murdoch I Conic Projection

Classification

Conic

Identifier

`murdoch1`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles.

Parallels: Equally spaced concentric circular arcs.

Poles: Arcs, one of which might become a point in the limit.

Symmetry: About any meridian.

Features

This is an equidistant projection that is nearly minimum-error. Scale is true along any meridian and is constant along any parallel. Scale is also true along two standard parallels. These must be calculated, however (see remark on parallels below). The total area of the mapped area is correct, but it is not equal-area everywhere.

Parallels

The parallels for this projection are not standard parallels, but rather limiting parallels. The special feature of this map, correct total area, holds between these parallels. The default parallels are [15 75].

Remarks

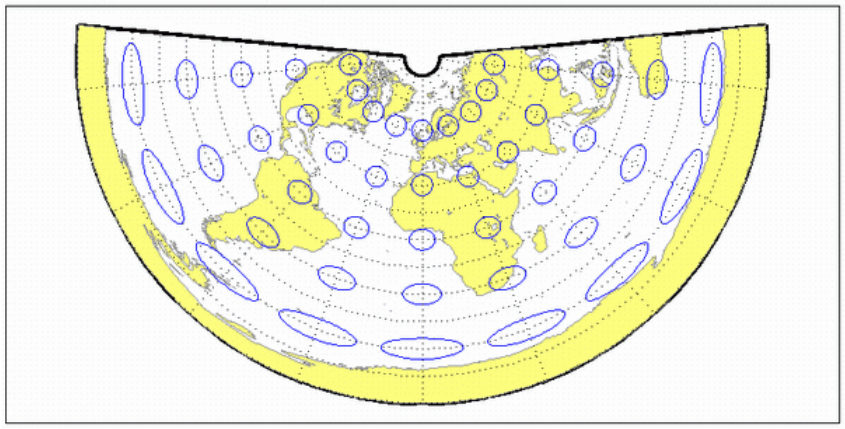
- Described by Patrick Murdoch in 1758.
- This implementation of the Murdoch I conic projection is applicable only for coordinates that are referenced to a sphere.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('murdoch1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

murdoch3

Murdoch III Minimum Error Conic Projection

Classification

Conic

Identifier

`murdoch3`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles.

Parallels: Equally spaced concentric circular arcs.

Poles: Arcs, one of which might become a point in the limit.

Symmetry: About any meridian.

Features

This is an equidistant projection that is minimum-error. Scale is true along any meridian and is constant along any parallel. Scale is also true along two standard parallels. These must be calculated, however (see remark on parallels below). The total area of the mapped area is correct, but it is not equal-area everywhere.

Parallels

The parallels for this projection are not standard parallels, but rather limiting parallels. The special feature of this map, correct total area, holds between these parallels. The default parallels are [15 75].

Remarks

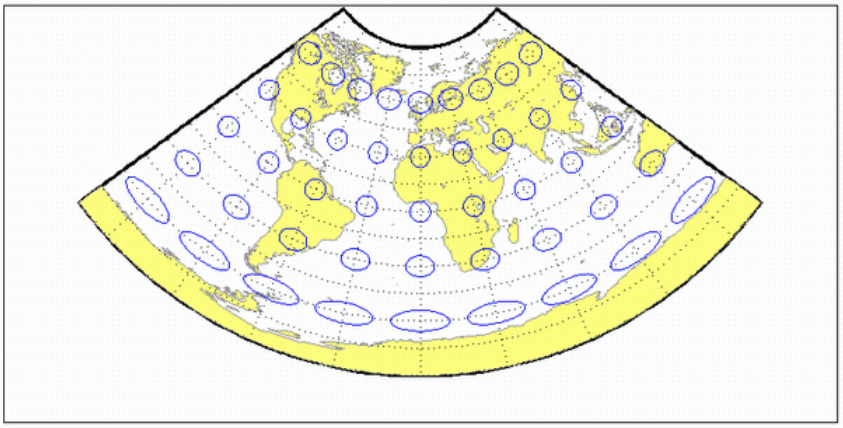
- Described by Patrick Murdoch in 1758, with errors corrected by Everett in 1904.
- This implementation of the Murdoch III minimum error conic projection is applicable only for coordinates that are referenced to a sphere.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('murdoch3', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

ortho

Orthographic Projection

Classification

Azimuthal

Identifier

`ortho`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown.

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from an infinite distance (that is, orthogonally). The center point is a pole in the common polar aspect, but can be any point. This projection has two significant properties. It looks like a globe, providing views of the Earth resembling those seen from outer space. Additionally, all great and small circles are either straight lines or elliptical arcs on this projection. Scale is true only at the center point and is constant in the circumferential direction along any circle having the center point as its center. Distortion increases rapidly away from the center point, the only place that is distortion-free. This projection is neither conformal nor equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

- This projection appears to have been developed by the Egyptians and Greeks by the second century B.C.
- This implementation of the orthographic projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid,

using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the orthographic projection using the ESRI authority code 102037. For example: `projcrs(102037,'Authority','ESRI')`.

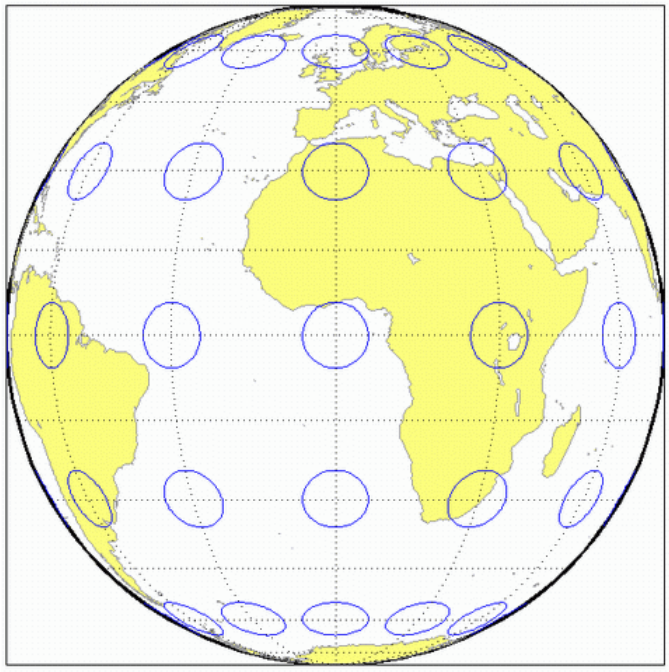
- Mapping Toolbox uses a different implementation of the orthographic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data greater than 89º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('ortho', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

pcarree

Plate Carree Projection

Classification

Cylindrical

Identifier

`pcarree`

Graticule

Meridians: Equally spaced straight parallel lines half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having the same spacing as the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the Equator. Distortion of both shape and area increases with distance from the Equator. Scale is true along all meridians (i.e., it is equidistant) and the Equator and is constant along any parallel and along the parallel of opposite sign.

Parallels

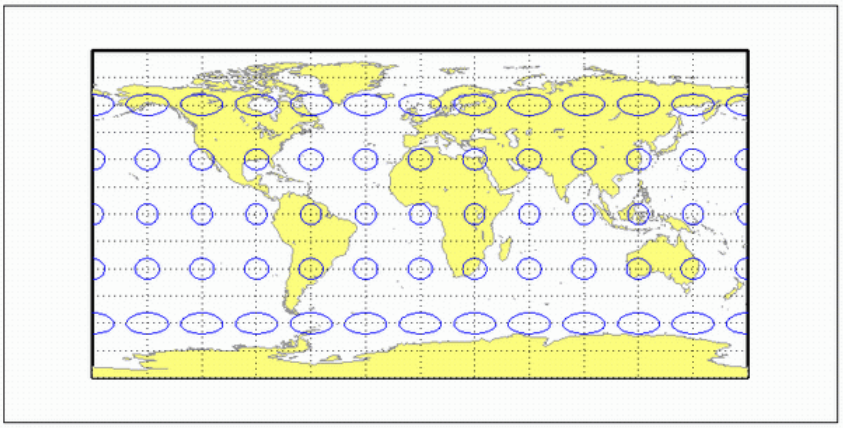
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

- This projection, like the more general Equidistant Cylindrical, is credited to Marinus of Tyre, thought to have invented it about A.D. 100. It may, in fact, have been originated by Erastosthenes, who lived approximately 275–195 B.C. The Plate Carrée has the most simply constructed graticule of any projection. It was used frequently in the 15th and 16th centuries and is quite common today in very simple computer mapping programs. It is the simplest and limiting form of the Equidistant Cylindrical projection. Another name for the Plate Carrée projection is the Simple Cylindrical. Its transverse aspect is the Cassini projection.
- On the sphere, this projection can have an arbitrary, oblique aspect, as controlled by the `Origin` property of the `axesm`-based map. On the ellipsoid, only the equatorial aspect is supported.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('pcarree', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

polycon

Polyconic Projection

Classification

Polyconic

Identifier

`polycon`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About the Equator or the central meridian.

Features

For this projection, each parallel has a curvature identical to its curvature on a cone tangent at that latitude. Since each parallel has its own cone, this is a “polyconic” projection. Scale is true along the central meridian and along each parallel. This projection is free of distortion only along the central meridian; distortion can be severe at extreme longitudes. This projection is neither conformal nor equal-area.

Parallels

By definition, this projection has no standard parallels, since every parallel is a *standard parallel*.

Remarks

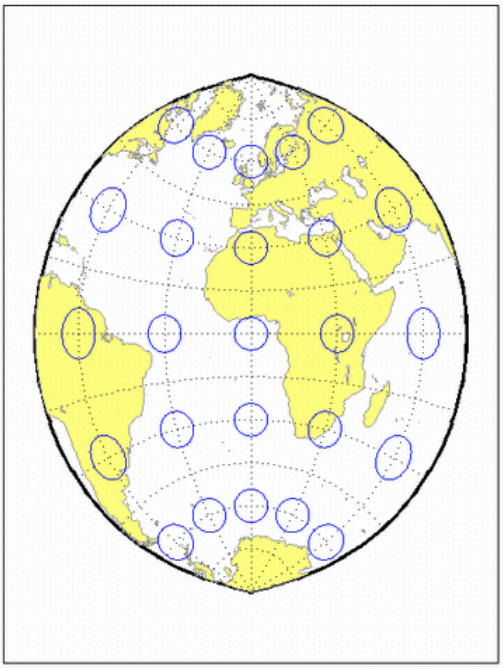
This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.

Limitations

Longitude data greater than 75º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('polycon', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`polyconstd` on page 10-113

Version History

Introduced before R2006a

polyconstd

Polyconic Projection — Standard

Classification

Polyconic

Identifier

`polyconstd`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About the Equator or the central meridian.

Features

`polyconstd` implements the Polyconic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `polycon` on page 10-111 for an alternative implementation based on rotating the rectifying sphere.

For this projection, each parallel has a curvature identical to its curvature on a cone tangent at that latitude. Since each parallel has its own cone, this is a “polyconic” projection. Scale is true along the central meridian and along each parallel. This projection is free of distortion only along the central meridian; distortion can be severe at extreme longitudes. This projection is neither conformal nor equal-area.

Parallels

By definition, this projection has no standard parallels, since every parallel is a *standard parallel*.

Remarks

- This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.

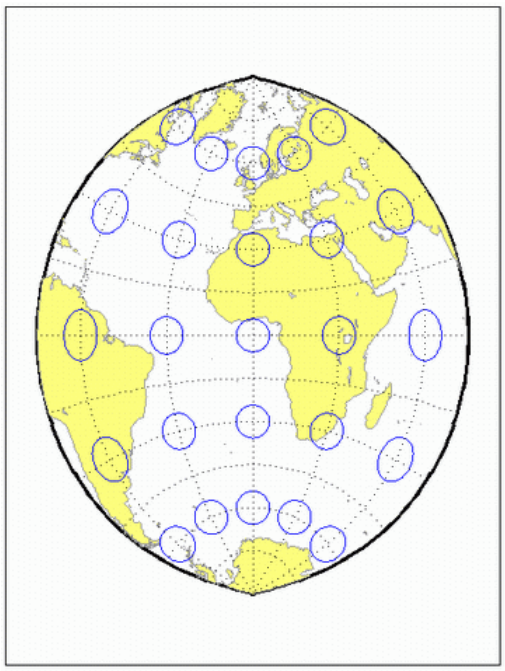
- Mapping Toolbox uses a different implementation of the standard polyconic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Longitude data greater than 75º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('polyconstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`polycon` on page 10-111

Version History

Introduced before R2006a

putnins5

Putnins P5 Projection

Classification

Pseudocylindrical

Identifier

`putnins5`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced portions of hyperbolas intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 21º14' parallels and is constant along any parallel, between any pair of parallels equidistant from the Equator, and along the central meridian. It is not free of distortion at any point. This projection is not equal-area, conformal, or equidistant.

Parallels

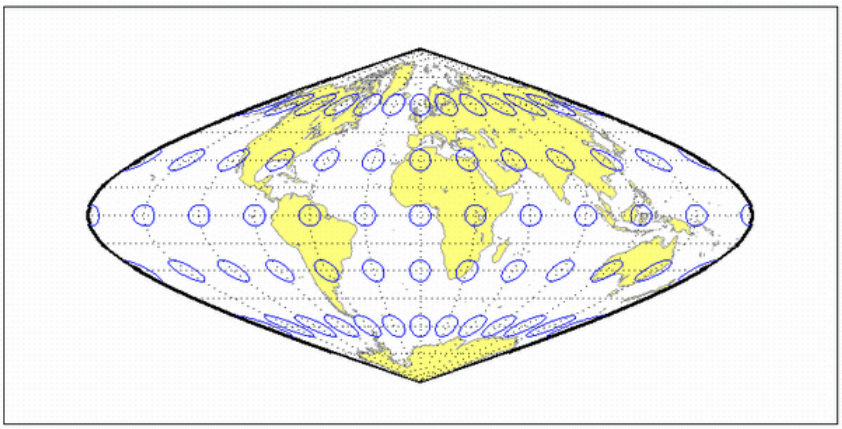
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 21º14'.

Remarks

- This projection was presented by Reinholds V. Putnins in 1934.
- This implementation of the Putnins P5 projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('putnins5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

quartic

Quartic Authalic Projection

Classification

Pseudocylindrical

Identifier

`quartic`

Graticule

Central Meridian: Straight line 0.45 as long as the Equator.

Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes gradually and is greatest near the Equator.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the Equator and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the Sinusoidal projection. It is free of distortion along the Equator. This projection is not conformal or equidistant.

Parallels

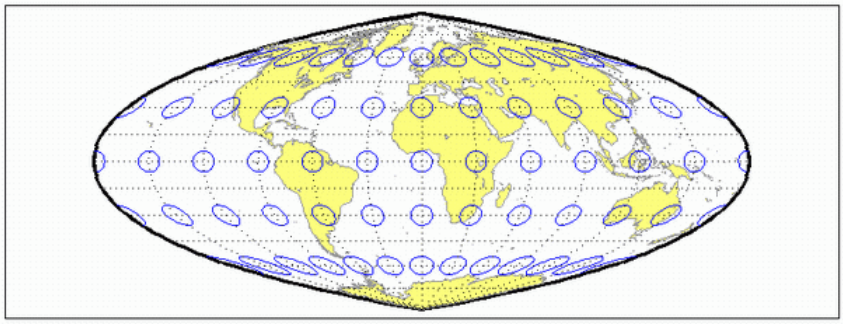
This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

This projection was presented by Karl Siemon in 1937 and independently by Oscar Sherman Adams in 1945.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('quartic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

robinson

Robinson Projection

Classification

Pseudocylindrical

Identifier

`robinson`

Graticule

Central Meridian: Straight line 0.51 as long as the Equator.

Other Meridians: Equally spaced, resemble elliptical arcs and are concave toward the central meridian.

Parallels: Straight parallel lines, perpendicular to the central meridian. Spacing is equal between the 38º parallels, decreasing outside these limits.

Poles: Lines 0.53 as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 38º parallels and is constant along any parallel or between any pair of parallels equidistant from the Equator. It is not free of distortion at any point, but distortion is very low within about 45º of the center and along the Equator. This projection is not equal-area, conformal, or equidistant; however, it is considered to *look right* for world maps, and hence is widely used by Rand McNally, the National Geographic Society, and others. This feature is achieved through the use of tabular coordinates rather than mathematical formulae for the graticules.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 38º.

Remarks

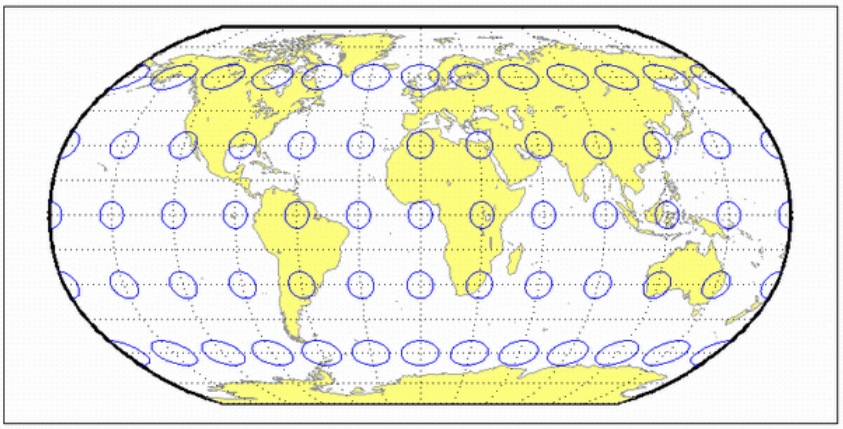
- This projection was presented by Arthur H. Robinson in 1963, and is also called the Orthophanic projection, which means *right appearing*.
- This implementation of the Robinson projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map

projection structure. You can create a `projcrs` object for the Robinson projection using the ESRI authority code 54030. For example: `projcrs(54030,'Authority','ESRI')`.

- Mapping Toolbox uses a different implementation of the Robinson projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('robinson', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

sinusoid

Sinusoidal projection

Classification

Pseudocylindrical

Identifier

`sinusoid`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This projection is equal-area. Scale is true along every parallel and along the central meridian. There is no distortion along the Equator or along the central meridian, but it becomes severe near the outer meridians at high latitudes.

Parallels

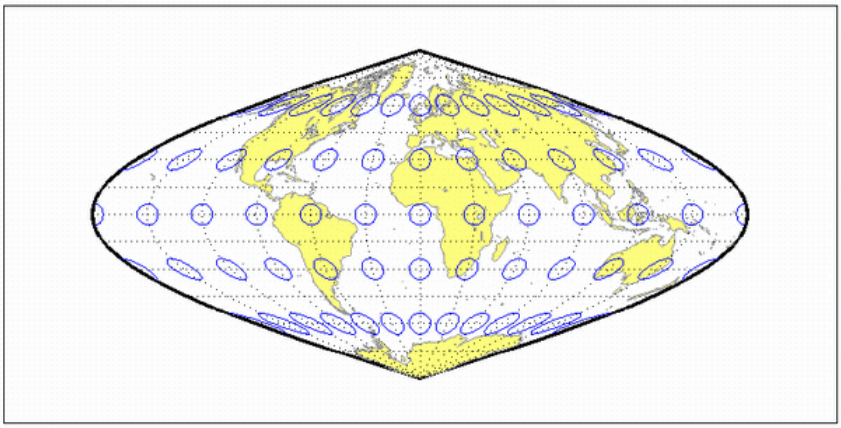
This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

- This projection was developed in the 16th century. It was used by Jean Cossin in 1570 and by Jodocus Hondius in Mercator atlases of the early 17th century. It is the oldest pseudocylindrical projection currently in use, and is sometimes called the Sanson-Flamsteed or the Mercator Equal-Area projection.
- Mapping Toolbox uses a different implementation of the sinusoidal projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('sinusoid', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

stereo

Stereographic Projection

Classification

Azimuthal

Identifier

`stereo`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing increases gradually away from this pole.

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from the point antipodal to the center point. The center point is a pole in the common polar aspect, but can be any point. This projection has two significant properties. It is conformal, being free from angular distortion. Additionally, all great and small circles are either straight lines or circular arcs on this projection. Scale is true only at the center point and is constant along any circle having the center point as its center. This projection is not equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

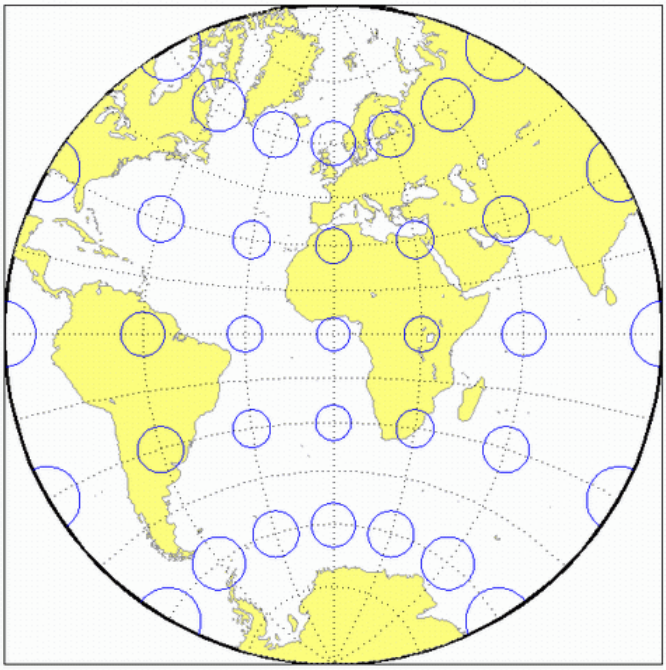
- The polar aspect of this projection appears to have been developed by the Egyptians and Greeks by the second century B.C.
- Mapping Toolbox uses a different implementation of the stereographic projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data greater than 90º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('stereo', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

modsine

Tissot Modified Sinusoidal Projection

Classification

Pseudocylindrical

Identifier

`modsine`

Graticule

Meridians: Sine curves converging at the Poles.

Parallels: Equally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and the central meridian

Features

This is an equal-area projection. Scale is constant along any parallel or any pair of equidistant parallels, and along the central meridian. It is not equidistant or conformal.

Parallels

There are no standard parallels for this projection.

Remarks

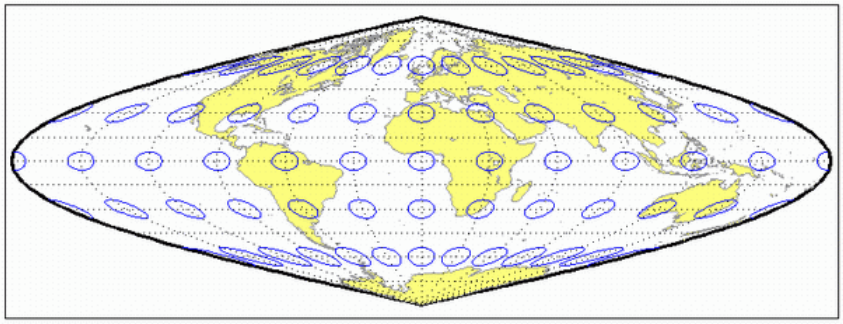
This projection was first described by N. A. Tissot in 1881

Limitations

This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('modsine', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

tranmerc

Transverse Mercator Projection

Classification

Cylindrical

Identifier

`tranmerc`

Features

This conformal projection is the transverse form of the Mercator projection and is also known as the Gauss-Krueger projection. It is not equal area, equidistant, or perspective.

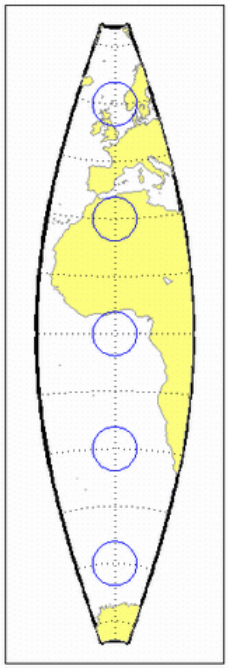
The scale is constant along the central meridian, and increases to the east and west. The scale at the central meridian can be set true to scale, or reduced slightly to render the mean scale of the overall map more correctly.

Remarks

- The uniformity of scale along its central meridian makes transverse Mercator an excellent choice for mapping areas that are elongated north-to-south. Its best known application is the definition of Universal Transverse Mercator (UTM) coordinates. Each UTM zone spans only 6 degrees of longitude, but the northern half extends from the equator all the way to 84 degrees north and the southern half extends from 80 degrees south to the equator. Other map grids based on transverse Mercator include many of the state plane zones in the U.S. and the U.K. National Grid.
- Mapping Toolbox uses a different implementation of the transverse Mercator projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('tranmerc', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

trystan

Trystan Edwards Cylindrical Projection

Classification

Cylindrical

Identifier

`trystan`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 37º24' parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

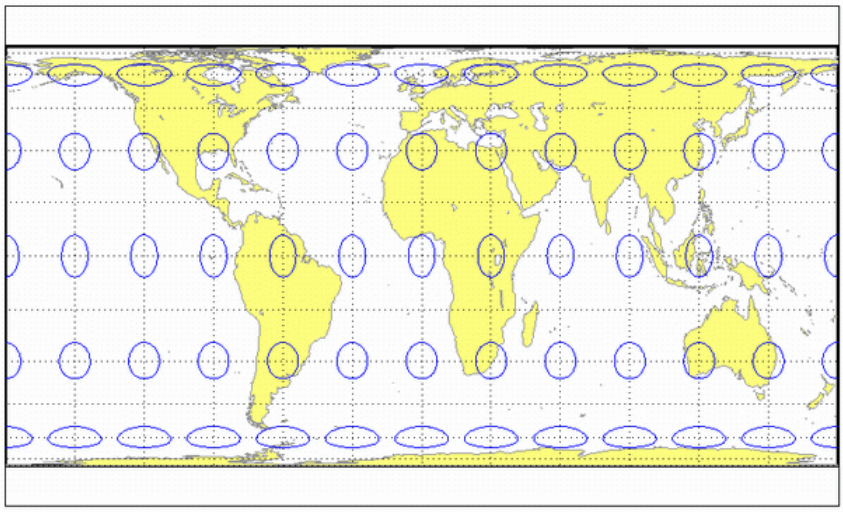
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 37º24'.

Remarks

This projection is named for Trystan Edwards, who presented it in 1953. It is a special form of the Equal-Area Cylindrical projection secant at 37º24'N and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('trystan', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

ups

Universal Polar Stereographic System

Classification

Azimuthal

Identifier

`ups`

Graticule

The graticule described is for the southern zone.

Meridians: Equally spaced straight lines centered on the South Pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the South Pole. Spacing increases gradually away from the circle of true scale along latitude 87 degrees, 7 minutes N. The opposite pole cannot be shown.

Poles: The South Pole is a point. The North Pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent to either the North or South Pole. It is conformal, being free from angular distortion. Additionally, all great and small circles are either straight lines or circular arcs on this projection. Scale is true along latitudes 87 degrees, 7 minutes N or S, and is constant along any other parallel. This projection is not equal area.

Parallels

The parallels 87 degrees, 7 minutes N and S are lines of true scale by virtue of the scale factor. There are no standard parallels for azimuthal projections.

Remarks

- This projection is a special case of the stereographic projection in the polar aspect. It is used as part of the Universal Transverse Mercator (UTM) system to extend coverage to the poles. This projection has two zones: “North” for latitudes 84º N to 90º N, and “South” for latitudes 80º S to 90º S. The defaults for this projection are: scale factor is 0.994, false easting and northing are 2,000,000 meters. The international ellipsoid in units of meters is used as the geoid model.

- Mapping Toolbox uses a different implementation of the UPS projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Version History

Introduced in R2006a

utm

Universal Transverse Mercator System

Classification

Cylindrical

Identifier

`utm`

Graticule

Meridians: Complex curves concave toward the central meridian.

Parallels: Complex curves concave toward the nearest pole.

Poles: Not shown.

Symmetry: About the central meridian or the Equator.

Features

This is a conformal projection with parameters chosen to minimize distortion over a defined set of small areas. It is not equal area, equidistant, or perspective. Scale is true along two straight lines on the map approximately 180 kilometers east and west of the central meridian, and is constant along other straight lines equidistant from the central meridian. Scale is less than true between, and greater than true outside the lines of true scale.

Parallels

There are no standard parallels for this projection. There are two lines of zero distortion by virtue of the scale factor.

Remarks

The UTM system divides the world between 80º S and 84º degrees N into a set of quadrangles called zones. Zones generally cover 6 degrees of longitude and 8 degrees of latitude. Each zone has a set of defined projection parameters, including central meridian, false eastings and northings and the reference ellipsoid. The projection equations are the Gauss-Krüger versions of the Transverse Mercator. The projected coordinates form a grid system, in which a location is specified by the zone, easting and northing.

The UTM system was introduced in the 1940s by the U.S. Army. It is widely used in topographic and military mapping.

Version History

Introduced in R2006a

vgrint1

Van der Grinten I Projection

Classification

Polyconic

Identifier

`vgrint1`

Graticule

Central Meridian: A straight line.

Meridians: Circular curves spaced equally along the equator and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are circular arcs concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator or the central meridian.

Features

In this projection, the world is enclosed in a circle. Scale is true along the Equator and increases rapidly away from the Equator. Area distortion is extreme near the poles. This projection is neither conformal nor equal-area.

Parallels

There are no standard parallels for this projection.

Remarks

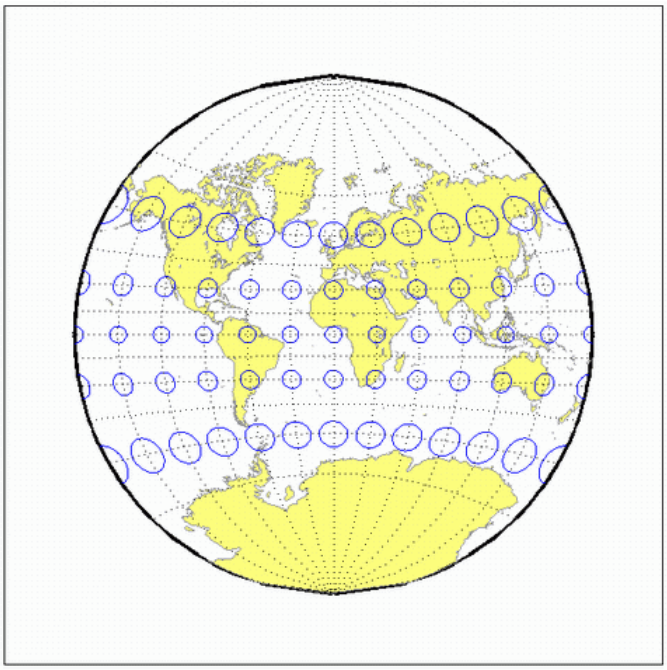
- This projection was presented by Alphons J. Van der Grinten in 1898. He obtained a U.S. patent for it in 1904. It is also known simply as the Van der Grinten projection (without the “I”).
- Mapping Toolbox uses a different implementation of the Van der Grinten I projection for displaying coordinates on `axesm`-based maps than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.
- The implementation of the Van der Grinten I projection for displaying coordinates on `axesm`-based maps is applicable only for coordinates that are referenced to a sphere. The implementation of the Van der Grinten I projection for projecting coordinates using the `projfwd` or `projinv` function is applicable for coordinates referenced to either a sphere or an ellipsoid.

Limitations

This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('vgrint1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

vperspec

Vertical Perspective Azimuthal Projection

Classification

Azimuthal

Identifier

`vperspec`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown, nor can distant parts of the closer hemisphere. The limit of visibility depends on the observation altitude.

Poles: The central pole is a point. The other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from a finite distance. Scale is true only at the center point, and is constant in the circumferential direction along any circle having the center point as its center. Distortion increases rapidly away from the center point, the only point which is distortion free. This projection is neither conformal nor equal area.

Remarks

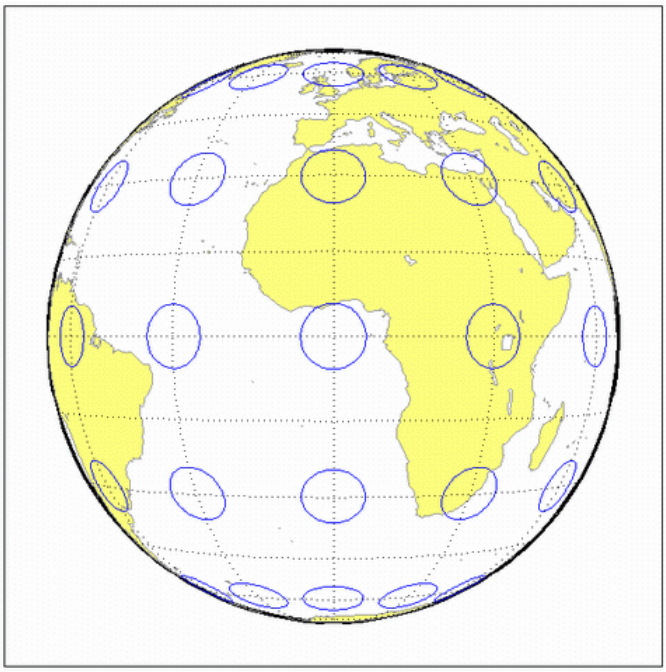
- This projection provides views of the globe resembling those seen from a spacecraft in orbit. The Orthographic projection is a limiting form with the observer at an infinite distance.
- This projection requires a view altitude parameter, which specifies the observer's altitude above the origin point. Because this parameter is unique to this projection and because the projection does not need any standard parallels, a special workaround is used. Rather than add an extra `axesm`-based map property just for `vperspec`, the `MapParallels` property is repurposed instead. You should assign the desired view altitude value to the `MapParallels` property. Provide a scalar value for length in the same units as the earth radius or semi-major axis length used in the `axesm`-based map reference ellipsoid (`'Geoid'`) property.
- This implementation of the vertical perspective azimuthal projection is applicable only for coordinates that are referenced to a sphere.

Limitations

Data more distant than the limit of visibility is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('vperspec', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

wagner4

Wagner IV Projection

Classification

Pseudocylindrical

Identifier

`wagner4`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced portions of ellipses concave toward the central meridian. The meridians 103º55' east and west of the central meridian are circular arcs.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 42º59' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is not as extreme near the outer meridians at high latitudes as for pointed-polar pseudocylindrical projections, but there is considerable distortion throughout the polar regions. It is free of distortion only at the two points where the 42º59' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 42º59'.

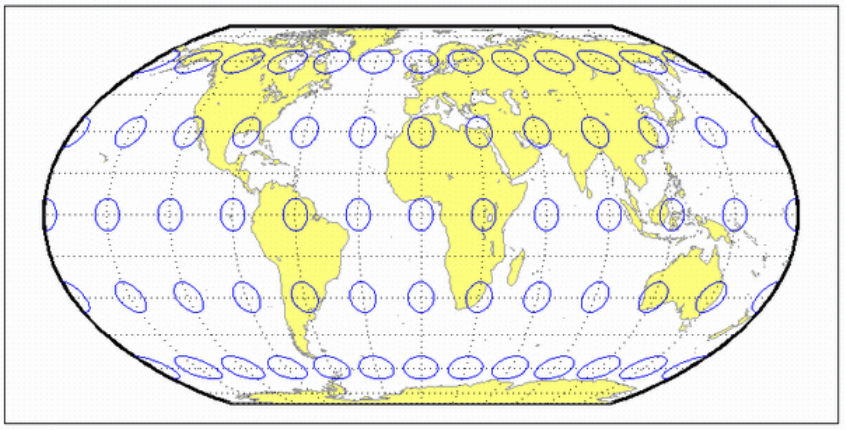
Remarks

This projection was presented by Karlheinz Wagner in 1932.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wagner4', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

werner

Werner Projection

Classification

Pseudoconic

Identifier

`werner`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves connecting points equally spaced along each parallel and concave toward the central meridian.

Parallels: Concentric circular arcs spaced at true distances along the central meridian, centered on one of the poles.

Poles: Points.

Symmetry: About the central meridian.

Features

This is an equal-area projection. It is a Bonne projection with one of the poles as its standard parallel. The central meridian is free of distortion. This projection is not conformal. Its heart shape gives it the additional descriptor *cordiform*.

Parallels

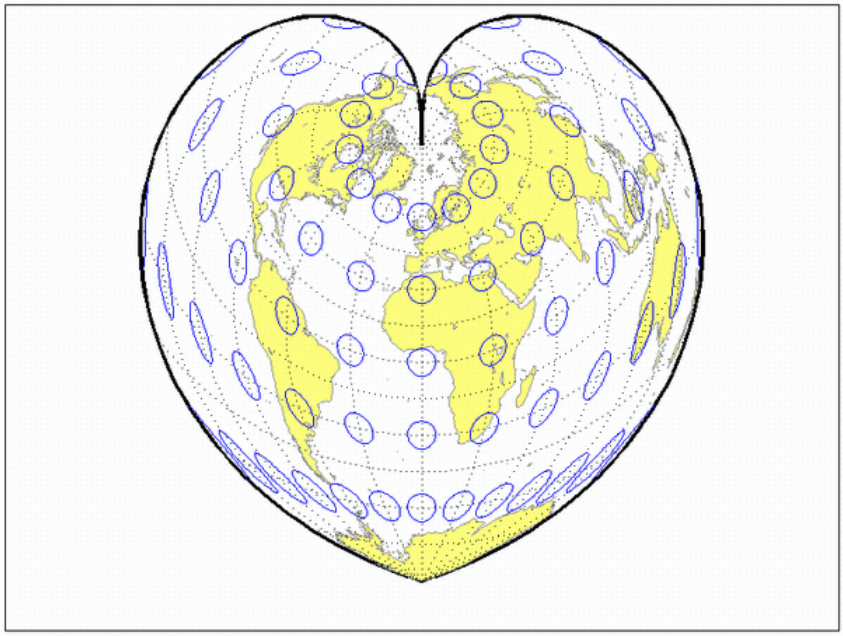
The standard parallel for this projection is set to 90º N.

Remarks

This projection was developed by Johannes Stabius (Stab) about 1500 and was promoted by Johannes Werner in 1514. It is also called the Stab-Werner projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('werner', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

wetch

Wetch Cylindrical Projection

Classification

Cylindrical

Identifier

`wetch`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90º from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a perspective projection from the center of the Earth onto a cylinder tangent to the central meridian. It is not equal-area, equidistant, or conformal. Scale is true along the central meridian and constant between two points equidistant in *x* and *y* from the central meridian. There is no distortion along the central meridian, but it increases rapidly away from the central meridian in the *y*-direction.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, which is the transverse aspect of the Central Cylindrical, the standard parallel *of the base projection* is by definition fixed at 0º.

Remarks

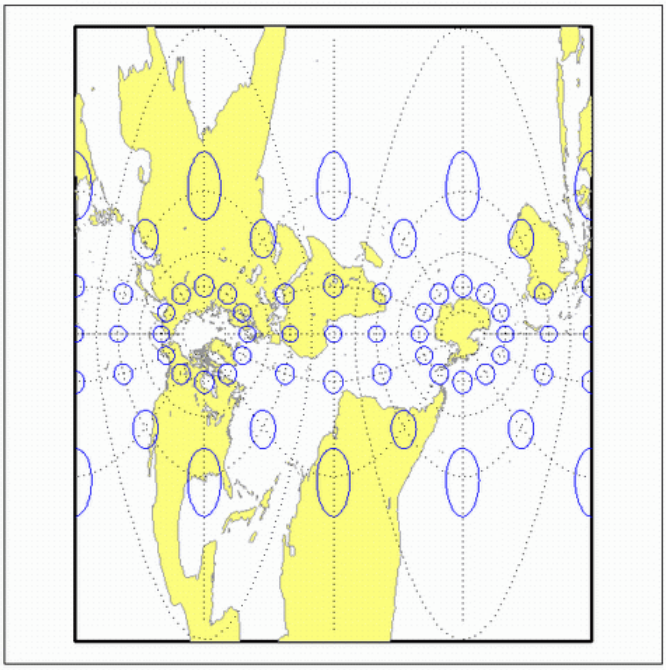
- This is the transverse aspect of the Central Cylindrical projection discussed by J. Wetch in the early 19th century.
- This implementation of the Wetch cylindrical projection is applicable only for coordinates that are referenced to a sphere.

Limitations

To prevent large *y*-values from dominating the display, data at *y*-values that would correspond to latitudes of greater than 75º in the normal aspect of the Central Cylindrical projection is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wetch', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

wiechel

Wiechel Projection

Classification

Pseudoazimuthal

Identifier

`wiechel`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced semicircles from pole to pole, concave toward the west.

Parallels: Concentric circles.

Pole: The central pole is a point; the other pole is a bounding circle.

Symmetry: Radially about the center point.

Features

This equal-area projection is a novelty map, usually centered at a pole, showing semicircular meridians in a pinwheel arrangement. Scale is correct along the meridians. This projection is not conformal.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

- This projection was presented by H. Wiechel in 1879.
- This implementation of the Wiechel projection is applicable only for coordinates that are referenced to a sphere.

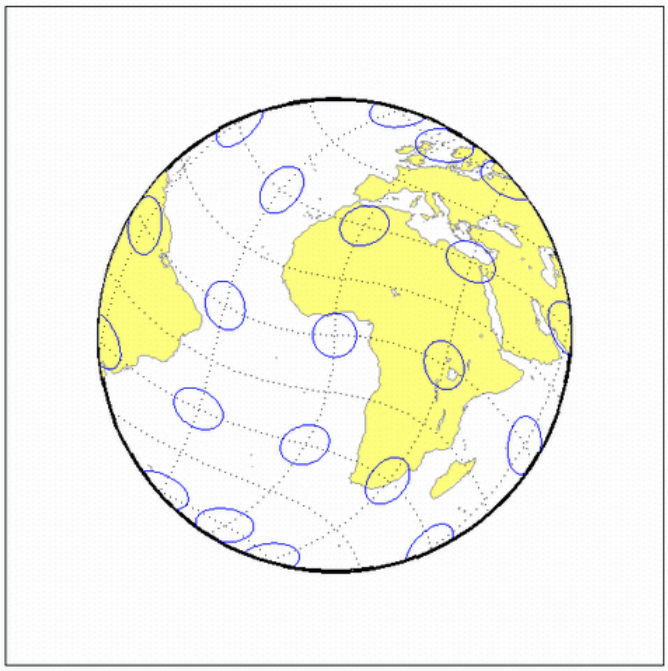
Limitations

Data greater than 65º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wiechel', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

winkel

Winkel I Projection

Classification

Pseudocylindrical

Identifier

`winkel`

Graticule

Central Meridian: Straight line at least half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines at least half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is an arithmetical average of the *x* and *y* coordinates of the Sinusoidal and Equidistant Cylindrical projections. Scale is true along the standard parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. There is no point free of distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

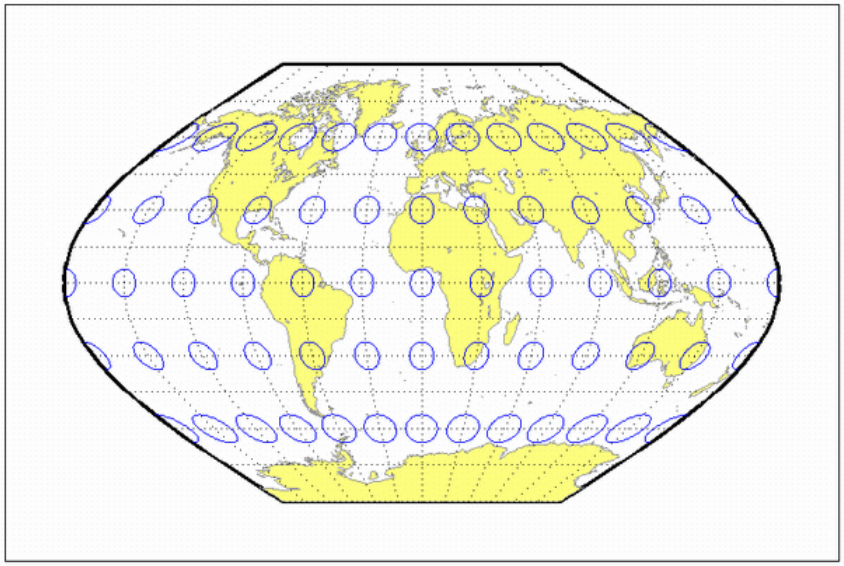
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. Any latitude may be chosen; the default is set to 50º28'.

Remarks

- This projection was developed by Oswald Winkel in 1914. Its limiting form is the Eckert V when a standard parallel of 0º is chosen.
- This implementation of the Winkel I projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Winkel I projection using the ESRI authority code 54018. For example: `projcrs(54018,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('winkel', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Version History

Introduced before R2006a

Applications

- “Build Pikes Peak RoadRunner 3D Scene Using RoadRunner HD Map” on page 11-2
- “Visualize Viewsheds and Coverage Maps Using Terrain” on page 11-9
- “Visualize Coverage Maps over Lunar Terrain Using Ray Tracing” on page 11-40
- “Radar Performance Analysis over Terrain” on page 11-58
- “Radar Vertical Coverage over Terrain” on page 11-68
- “Process Large GeoTIFF File as Blocked Image” on page 11-77
- “Crop and Mask Large GeoTIFF File Using Shapefile” on page 11-85
- “Hurricane Natural Catastrophe Risk Estimation” on page 11-92

Build Pikes Peak RoadRunner 3D Scene Using RoadRunner HD Map

This example shows how to build a RoadRunner 3D scene for Pikes Peak, Colorado, using MATLAB® functions. You can build a scene for the full road, or a section of Pikes Peak Road using a Keyhole Markup Language (KML) file containing its latitude-longitude coordinates and a Geostationary Earth Orbit Tagged Image File Format (GeoTIFF) file containing its elevation. Use Mapping Toolbox™ to import the data files for the road.

Import KML File for Pikes Peak

You can choose to build the RoadRunner HD Map for either a section of the road or the full road. Note that building a section of the road requires significantly less time than building the entire scene. In this example, you import the coordinates for the road centers into MATLAB from a KML file (Map data ©2022 Google), and then plot the data imported from the KML file to view the coordinates.

To build only a section of the road, specify `buildFullRoad` as `false`. To build the full road, specify it as `true`.

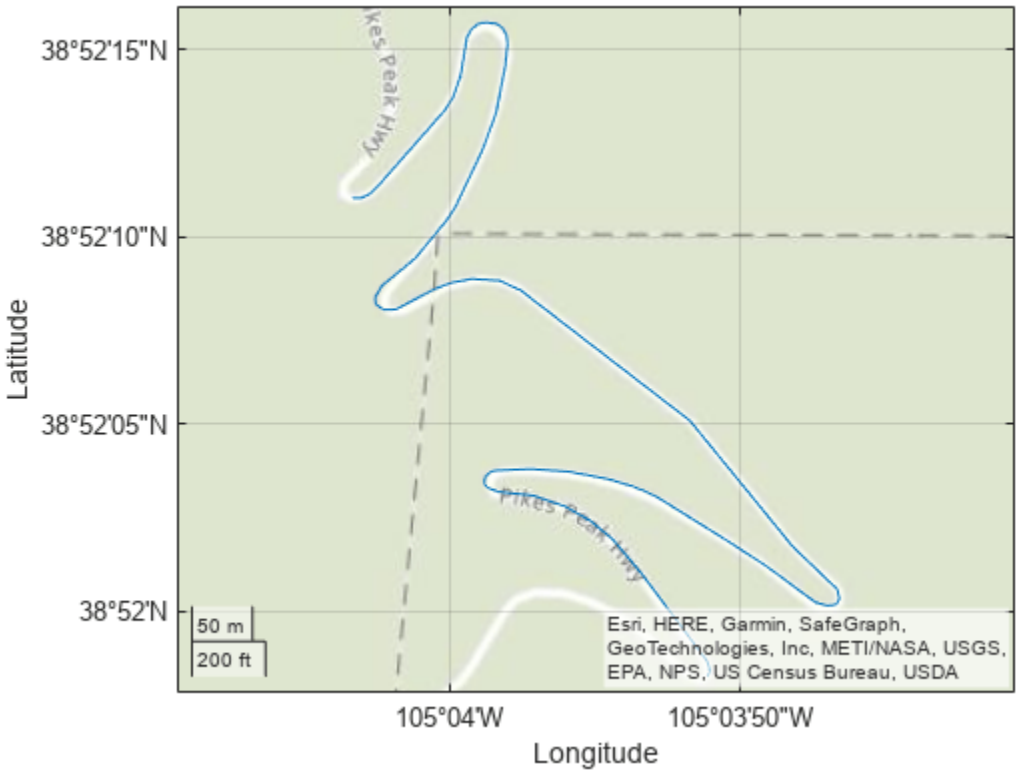
```
buildFullRoad = false;
```

Read the data from the KML file.

```
if buildFullRoad
    kmlData = readgeotable("PikesPeakFullRoadData.kml");
else
    kmlData = readgeotable("PikesPeakRoadSectionData.kml");
end
```

Plot the coordinates of Pikes Peak road.

```
geoplot(kmlData)
geobasemap topographic
```



Import GeoTIFF File for Pikes Peak Elevation

To include elevation for Pikes Peak Road, download terrain data for the area from the United States Geological Survey National Map Services, using this link (United States Geological Survey National Map Services TNM Download). This data has been processed by cropping the USGS data to the Pikes Peak area.

Convert the geospatial table to a table of road centers to obtain the latitude and longitude coordinates for the road.

```
T = geotable2table(kmlData,["Latitude","Longitude"]);
[lat,lon] = polyjoin(T.Latitude,T.Longitude);
```

Query the elevations of the road coordinates from the terrain data.

```
filename = "PikesPeak.tif";
outputType = "OutputType";
type = "double";
[georefGrid,spatialRef] = readgeoraster(filename,outputType,type);
ctrElev = geointerp(georefGrid,spatialRef,lat,lon);
```

Create RoadRunner HD Map

Create the RoadRunner HD Map and set the geographic reference origin.

Create an empty RoadRunner HD Map.

```
rrMap = roadrunnerHDMap;
```

Compute the georeference origin as the center of the bounding quadrangle.

```
[latlim,lonlim] = geoquadline(lat,lon);
lat0 = mean(latlim);
lon0 = mean(lonlim);
```

Set the geographic reference for the region of interest.

```
rrMap.GeoReference = [lat0 lon0];
```

Upsample Road Data

Because the data points obtained from the KML file are sparse, and the road contains sharp curves, you must upsample the data to avoid modeling inaccurate road lanes. You can upsample the data by using `helperRoadDimensions` helper function. First, you must transform the imported latitude and longitude coordinates to *xy*-map coordinates using a projected CRS. Then, using the *xy*-map coordinates for the centers of the road, you must set the road width. In this example, Pikes Peak Road is a two-lane road, so you can assume a total road width of 7.4 meters based on the average lane width of 3.7 meters.

Read the Transverse Mercator projected CRS from the RoadRunner HD Map.

```
p = readCRS(rrMap);
```

Project the latitude and longitude coordinates to *xy*-coordinates.

```
[x,y] = projfwd(p,lat,lon);
```

Define the road centers and the road width.

```
rdCtrs = [x y ctrElev];
rdWidth = 7.4;
```

Upsample the data for the lane boundaries to obtain a better result for the road curves.

```
[lftBndry,rgtBndry,ctrBndry] = helperRoadDimensions(rdCtrs,rdWidth);
```

Create Road Network

Create the RoadRunner HD Map road network using the interpolated data, and modify the road to resemble Pikes Peak Road, which consists of two lanes and three lane boundaries. Then, apply a solid white marking to the outer lane boundaries and a double yellow marking between the two lanes for the entire length of the road.

To improve performance as the number of objects in the map increases, initialize the `Lanes` and `LaneBoundaries` properties of the HD Map.

```
rrMap.Lanes(2,1) = roadrunner.hdmap.Lane;
rrMap.LaneBoundaries(3,1) = roadrunner.hdmap.LaneBoundary;
```

Assign `Lane` property values. Use `deal` to match the input lists to the output lists.

```
[rrMap.Lanes.ID] = deal("Lane1","Lane2");
[rrMap.Lanes.TravelDirection] = deal("Backward","Forward");
[rrMap.Lanes.LaneType] = deal("Driving");
```

Average the coordinates of the left and right lane boundaries with the corresponding coordinates of the center lane boundary to obtain the lane centers.

```
lftLane = (lftBndry+ctrBndry)/2;
rgtLane = (rgtBndry+ctrBndry)/2;
[rrMap.Lanes.Geometry] = deal(lftLane,rgtLane);
```

Assign IDs and the corresponding coordinates to the three lane boundaries.

```
[rrMap.LaneBoundaries.ID] = deal("Left","Center","Right");
[rrMap.LaneBoundaries.Geometry] = deal(lftBndry,ctrBndry,rgtBndry);
```

Associate the lane boundaries to their lanes using their IDs. Note that both lanes share the center lane boundary.

```
leftBoundary(rrMap.Lanes(1),"Left",Alignment="Forward");
rightBoundary(rrMap.Lanes(1),"Center",Alignment="Forward");
leftBoundary(rrMap.Lanes(2),"Center",Alignment="Forward");
rightBoundary(rrMap.Lanes(2),"Right",Alignment="Forward");
```

Define the file path to the solid white lane marking asset.

```
solidWhiteAsset = roadrunner.hdmap.RelativeAssetPath(AssetPath="Assets/Markings/SolidSingleWhite
```

Define the path to the solid double yellow lane marking asset. Add both lane markings to the HD Map.

```
doubleYellowAsset = roadrunner.hdmap.RelativeAssetPath(AssetPath="Assets/Markings/SolidDoubleYel
rrMap.LaneMarkings(2,1) = roadrunner.hdmap.LaneMarking;
[rrMap.LaneMarkings.ID] = deal("SolidWhite","DoubleYellow");
[rrMap.LaneMarkings.AssetPath] = deal(solidWhiteAsset,doubleYellowAsset);
```

Specify that the markings span the entire length of their lane boundary. Create a reference and attribution for the solid double yellow marking.

```
markingSpan = [0 1];
markingRefSY = roadrunner.hdmap.MarkingReference(MarkingID=roadrunner.hdmap.Reference(ID="Double
markingAttribSY = roadrunner.hdmap.ParametricAttribution(MarkingReference=markingRefSY,Span=mark
```

Create a reference and attribution for the solid white marking. Then, apply the solid white marking to the left and right lane boundaries and the solid double yellow marking to the center lane boundary.

```
markingRefSW = roadrunner.hdmap.MarkingReference(MarkingID=roadrunner.hdmap.Reference(ID="SolidW
markingAttribSW = roadrunner.hdmap.ParametricAttribution(MarkingReference=markingRefSW,Span=mark
[rrMap.LaneBoundaries.ParametricAttributes] = deal(markingAttribSW,markingAttribSY,markingAttrib
```

Set Geographic Boundaries

Setting the geographic boundaries for the RoadRunner HD Map centers the scene on the imported road and enables you to insert the road into the scene without using the **World Settings Tool** in RoadRunner.

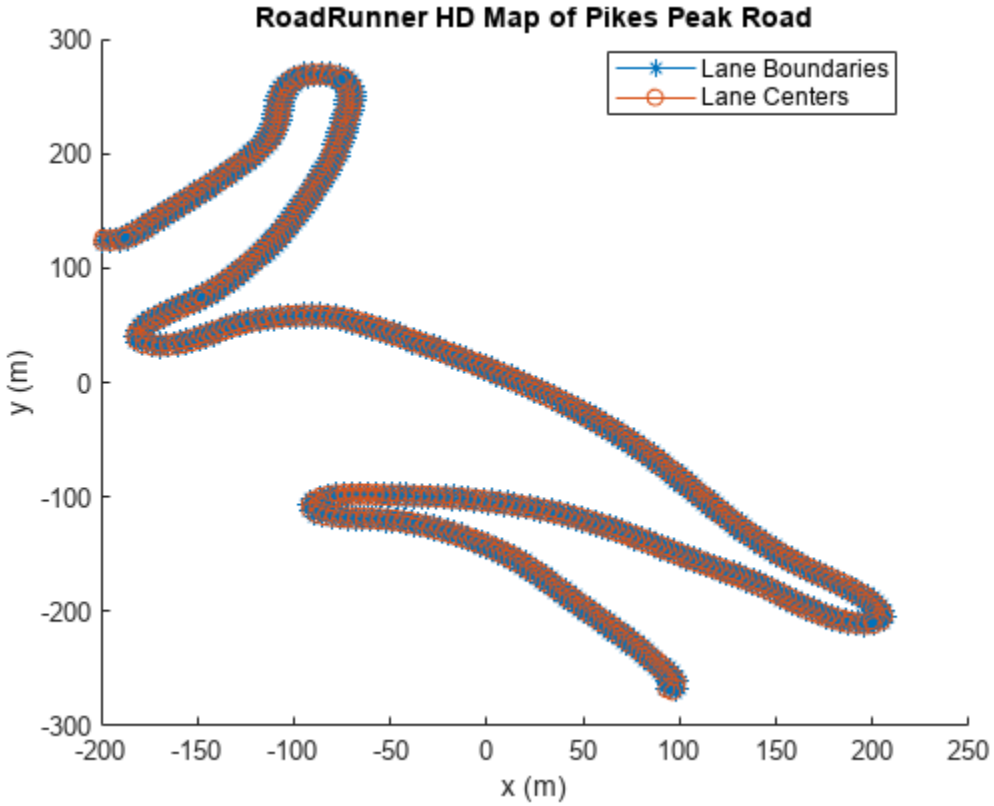
Set the geographic bounds for the map as the minimum and maximum coordinate values of the center boundary.

```
geoBounds = [min(ctrBndry); max(ctrBndry)];
rrMap.GeographicBoundary = geoBounds;
```

Plot the lane centers and lane boundaries.

```
plot(rrMap)
title('RoadRunner HD Map of Pikes Peak Road')
```

```
xlabel('x (m)')
ylabel('y (m)')
```



Export RoadRunner HD Map and Import to RoadRunner

Save the RoadRunner HD Map to a file. Import it into RoadRunner by adding the file to a folder in the **Library Browser** and dragging it into the scene. For more information on importing an HD Map file into RoadRunner, see “Import HD Map File into RoadRunner” (RoadRunner). Build the scene using the **Scene Builder Tool**.

Specify an output filename based on the data used to construct the RoadRunner HD Map.

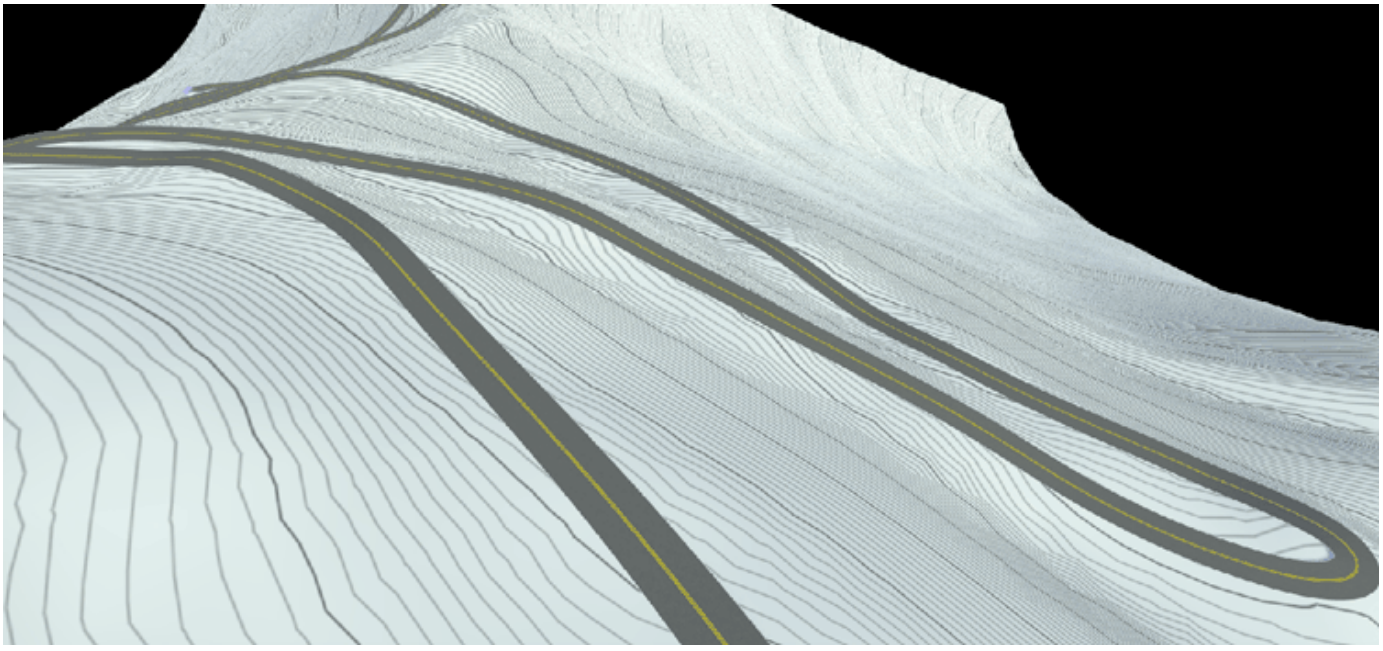
```
if buildFullRoad
    fileName = "PikesPeakFullRoad.rrhd";
else
    fileName = "PikesPeakRoadSection.rrhd";
end
```

Write the map to the file.

```
write(rrMap,fileName)
```

This figure shows a 3D scene of Pikes Peak Road built using RoadRunner Scene Builder.

To visualize the terrain, you can import the PikesPeak.tif file into RoadRunner using the Elevation Map Tool (RoadRunner).



See Also

Functions
`readgeotable` | `geoplot`

Objects
`projcrs` | `roadrunner` | `roadrunnerHDMap`

Visualize Viewsheds and Coverage Maps Using Terrain

Display viewsheds and coverage maps for an area of interest using satellite imagery and terrain data. A *viewshed* includes areas that are within line of sight (LOS) of a set of points, and a *coverage map* includes received power for the areas within range of a set of wireless transmitters. You can use a viewshed as a first-order approximation of a coverage map.

By combining the mapping capabilities in Mapping Toolbox™ with the RF propagation capabilities in Antenna Toolbox™, you can create contoured coverage maps and use the contour data for both visualization and analysis. This example shows you how to:

- View imagery and terrain on 2-D maps
- View imagery, terrain, and viewsheds on 3-D relief maps
- View coverage maps in 2-D using contour plots and bar graphs
- Determine whether nearby points are within coverage
- View coverage maps for other regions of interest

View Imagery on 2-D Map

You can view imagery for an area of interest by plotting data into geographic axes. Geographic axes enable you to plot 2-D data over basemaps, explore regions interactively, and add data tips to points of interest.

Specify the geographic coordinates and heights of two cell towers in the San Fernando Valley region. Specify the heights using meters above the terrain.

```
lat1 = 34.076389;
lon1 = -118.468944;
towerHeight1 = 17.1;
text1 = "Tower 1";

lat2 = 34.386389;
lon2 = -118.329778;
towerHeight2 = 30.5;
text2 = "Tower 2";
```

Display the locations of the cell towers over a satellite basemap. Throughout the example, create larger maps by increasing the sizes of the figures.

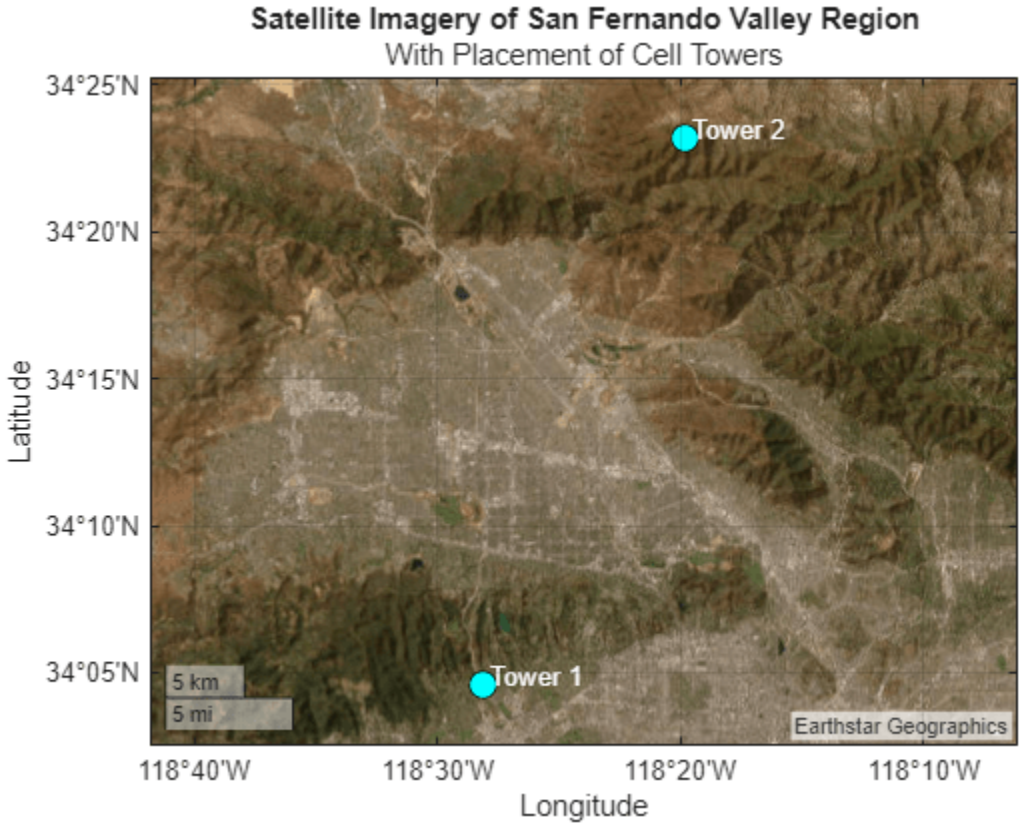
```
f1 = figure;
fSize = [560 420];
f1.Position(3:4) = fSize;

basemap = "satellite";
geobasemap(basemap)

geoplot([lat1 lat2],[lon1 lon2], ...
    LineStyle="none",Marker="o",MarkerSize=10, ...
    MarkerEdgeColor="k",MarkerFaceColor="c")
```

Prepare to create a 3-D relief map for the same region by querying the zoom level, latitude limits, and longitude limits of the geographic axes. To ensure that the relief map covers the same region, change the zoom level of the geographic axes to an integer.

```
gx = gca;
zoomLevel = floor(gx.ZoomLevel);
gx.ZoomLevel = zoomLevel;

[latlim,lonlim] = geolimits(gx);
```

Customize the map by labeling the cell towers and adding a title.

```
txtDelta = 0.005;
text(lat1 + txtDelta,lon1 + txtDelta,text1,Color="w")
text(lat2 + txtDelta,lon2 + txtDelta,text2,Color="w")
title("Satellite Imagery of San Fernando Valley Region")
subtitle("With Placement of Cell Towers")
```



View Terrain on 2-D Map

Visualize the terrain for the same area of interest by plotting imported elevation data into an `axesm`-based map. `axesm`-based maps enable you to display geographic data using a map projection.

Read Terrain Elevation Data

Read terrain elevation data for the area of interest from a Web Map Service (WMS) server. WMS servers provide geospatial data such as elevation, temperature, and weather from web-based sources.

Search the WMS Database for terrain elevation data hosted by MathWorks®. The `wmsfind` function returns search results as layer objects. A *layer* is a data set that contains a specific type of geographic information, in this case elevation data.

```
layer = wmsfind("mathworks",SearchField="serverurl");
layer = refine(layer,"elevation");
```

Read the elevation data from the layer into the workspace as an array and a spatial reference object in geographic coordinates. Get quantitative data by specifying the image format as BIL.

```
[Z,RZ] = wmsread(layer,Latlim=latlim,Lonlim=lonlim,ImageFormat="image/bil");
```

Prepare the elevation data for plotting by converting the data type to `double`.

```
Z = double(Z);
```

Find Tower Elevations

Calculate the terrain elevation at the base of each cell tower. The elevations are in meters above mean sea level.

```
height1 = geointerp(Z,RZ,lat1,lon1,"nearest")
```

```
height1 =
177
```

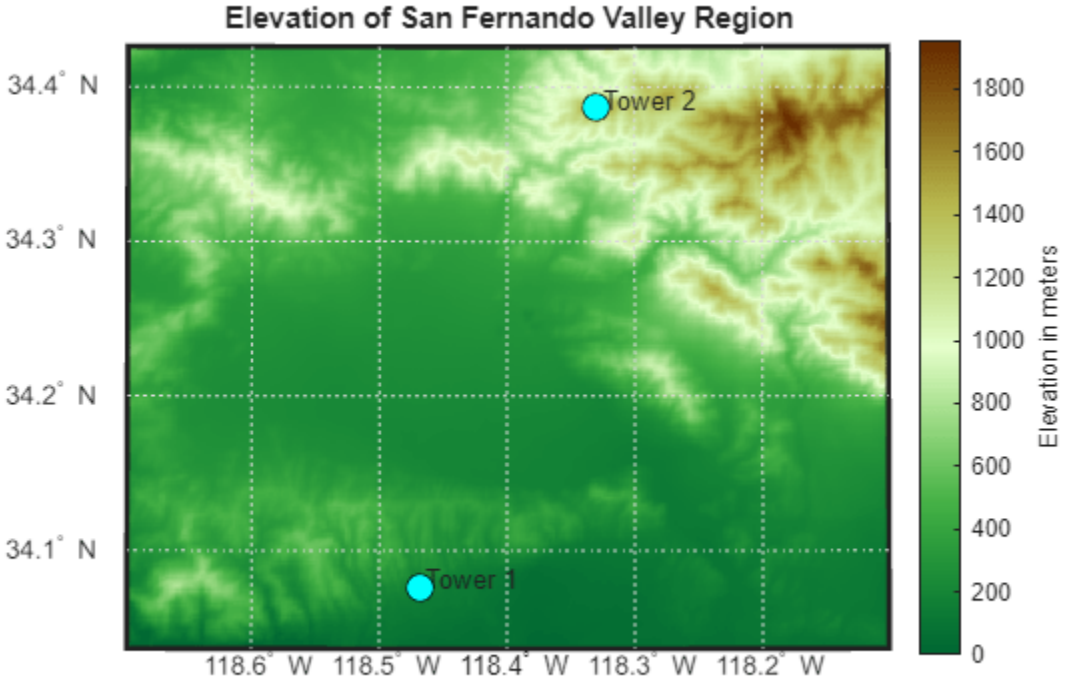
```
height2 = geointerp(Z,RZ,lat2,lon2,"nearest")
```

```
height2 =
1422
```

Visualize Elevation and Tower Locations

Create an `axesm`-based map with an appropriate projection for the region. Then, display the elevation data as a surface using an appropriate colormap. Note that the areas of highest elevation are in the northeast corner.

```
f2 = figure;
f2.Position(3:4) = fSize;

usamap(Z,RZ)
geoshow(Z,RZ,DisplayType="texturemap")
demcmap(Z,256)

geoshow(lat1,lon1,DisplayType="point",ZData=height1, ...
    MarkerEdgeColor="k",MarkerFaceColor="c",MarkerSize=10,Marker="o")
geoshow(lat2,lon2,DisplayType="point",ZData=height2, ...
     MarkerEdgeColor="k",MarkerFaceColor="c",MarkerSize=10,Marker="o")

textm(lat1 + txtDelta,lon1 + txtDelta,text1)
textm(lat2 + txtDelta,lon2 + txtDelta,text2)
title("Elevation of San Fernando Valley Region")

cb = colorbar;
cb.Label.String = "Elevation in meters";
```



View Imagery, Terrain, and Viewshed on 3-D Relief Map

Visualize the imagery, terrain, and viewshed for the region by plotting the terrain data over a basemap image in a standard axes. Basemap images enable you to provide geographic context for plot types that geographic axes do not support, such as 3-D surfaces.

Read Basemap Image

Read a basemap image using the same basemap, geographic limits, and zoom level as the geographic axes. The `readBasemapImage` function reads the image into the workspace as an array, a spatial reference object in Web Mercator (WGS 84/Pseudo-Mercator) coordinates, and an attribution string.

```
[A,RA,attrib] = readBasemapImage(basemap,latlim,lonlim,zoomLevel);
```

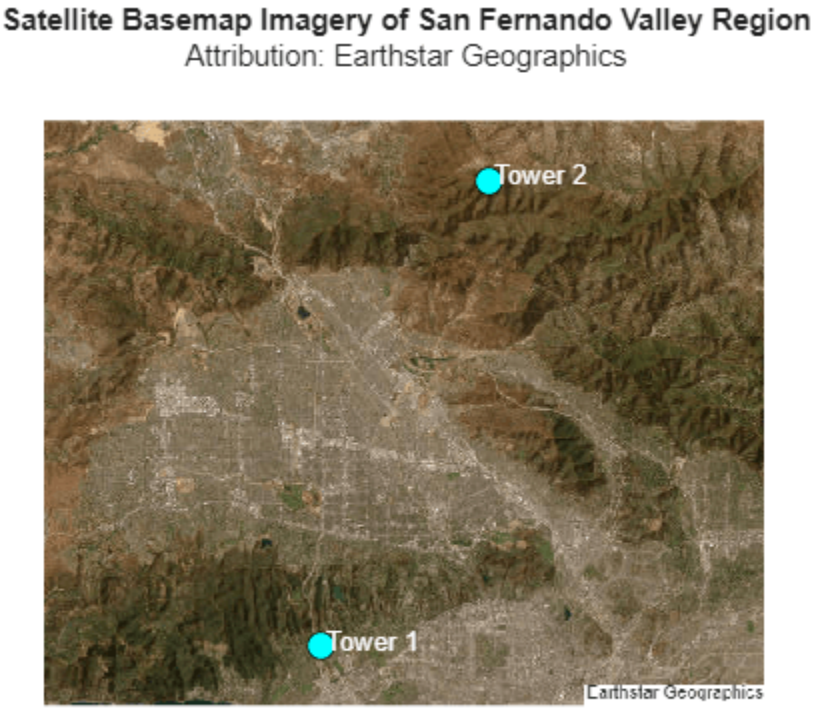
To display the cell tower locations over the basemap image, you must first project the geographic coordinates to Web Mercator coordinates.

```
proj = RA.ProjectedCRS;
[x1,y1] = projfwd(proj,lat1,lon1);
[x2,y2] = projfwd(proj,lat2,lon2);
```

Plot the projected coordinates over the basemap image. Note that this visualization is similar to the geographic axes visualization.

```
f3 = figure;
f3.Position(3:4) = fSize;
```

```
axis off
hold on

mapshow(A,RA)
plot([x1 x2],[y1 y2], ...
    LineStyle="none",Marker="o",MarkerSize=10, ...
    MarkerEdgeColor="k",MarkerFaceColor="c")

txtDeltaXY = 500;
text(x1 + txtDeltaXY,y1 + txtDeltaXY,text1,Color="w")
text(x2 + txtDeltaXY,y2 + txtDeltaXY,text2,Color="w")
title("Satellite Basemap Imagery of San Fernando Valley Region")
subtitle("Attribution: " + attrib)
```



Calculate Viewsheds from Towers

Read the terrain data again, this time using array dimensions and geographic limits that match the dimensions and limits of the basemap image.

```
[latlim,lonlim] = projinv(proj,RA.XWorldLimits,RA.YWorldLimits);
szA = RA.RasterSize;

[Z,RZ] = wmsread(layer,Latlim=latlim,Lonlim=lonlim, ...
    ImageHeight=szA(1),ImageWidth=szA(2),ImageFormat="image/bil");
Z = double(Z);
```

Calculate the viewshed for each tower by specifying the terrain data and the cell tower coordinates. The first argument returned by each `viewshed` call indicates visible areas using `1` values and obscured areas using `0` values.

```
[vis1,visR1] = viewshed(Z,RZ,lat1,lon1,towerHeight1);
[vis2,visR2] = viewshed(Z,RZ,lat2,lon2,towerHeight2);
```

Calculate the combined viewshed for both towers. To avoid plotting the obscured areas, replace the `0` values with `NaN` values.

```
visCombined = vis1 | vis2;
vis = visCombined;
vis = double(vis);
vis(vis == 0) = NaN;
visR = visR2;
```

Extract the geographic coordinates of the elevation data from its spatial reference object. Then, project the coordinates to Web Mercator coordinates.

```
[latz,lonz] = geographicGrid(RZ);
[xz,yz] = projfwd(proj,latz,lonz);
```

View Individual and Combined Viewsheds

Display the viewshed for each tower and the combined viewshed for both towers in a tiled layout.

Define a colormap for the viewshed. Use turquoise for visible areas and black for obscured areas.

```
cmapt = [0 0 0];
covcolor = [0.1034  0.8960  0.7150];
cmapt(2,:,:) = covcolor;
```

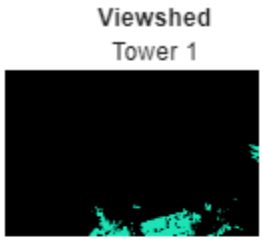
Create a tiled layout. Display the viewshed for the individual towers on the left.

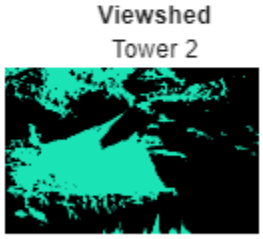
```
f4 = figure;
f4.Position(3:4) = fSize;

clim([0 1])
colormap(cmapt)
tiledlayout(2,3,Padding="tight",TileSpacing="tight")

nexttile
geoshow(vis1,visR1,EdgeColor="none",DisplayType="surface")
axis off
title("Viewshed")
subtitle("Tower 1")

nexttile(4)
geoshow(vis2,visR2,EdgeColor="none",DisplayType="surface")
axis off
title("Viewshed")
subtitle("Tower 2")
```

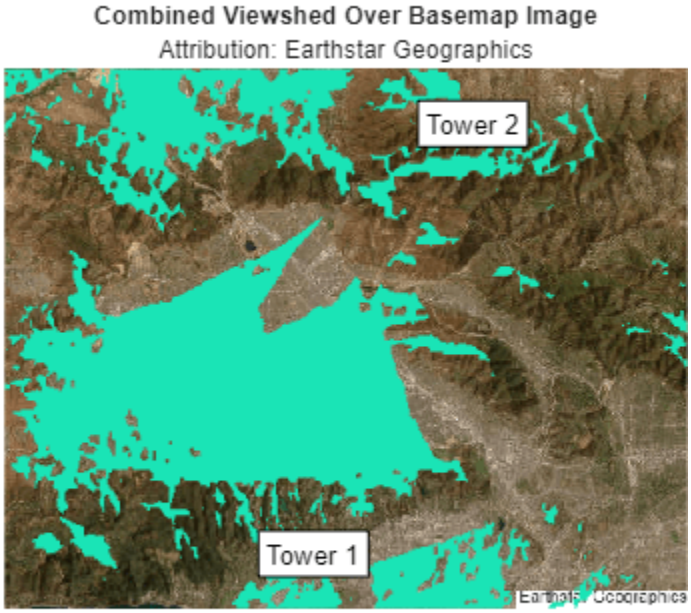
Display the basemap image and the combined viewshed on the right. Specify the *z*-coordinates for the labels using the terrain height (above mean sea level) and the tower height (above the terrain).

```
nexttile([2,2])
mapshow(A,RA)
```

```
hold on
mapshow(xz,yz,vis,EdgeColor="none",DisplayType="surface")
axis off

zdelta = 150;
z1 = height1 + towerHeight1 + zdelta;
z2 = height2 + towerHeight2 + zdelta;

text(x1,y1,z1,text1,Color="k",EdgeColor="k",BackgroundColor="w")
text(x2,y2,z2,text2,Color="k",EdgeColor="k",BackgroundColor="w")
title("Combined Viewshed Over Basemap Image")
subtitle("Attribution: " + attrib)
```







View Basemap Image and Viewshed on 3-D Relief Map

Prepare the combined viewshed for plotting.

- Find the indices of visible areas in the combined viewshed.
- Replace the visible areas in the combined viewshed with the terrain data (the corresponding values of Z).

```
[row,col] = find(visCombined);
sz = size(visCombined);
ind = sub2ind(sz,row,col);
vis(ind) = Z(ind);
```

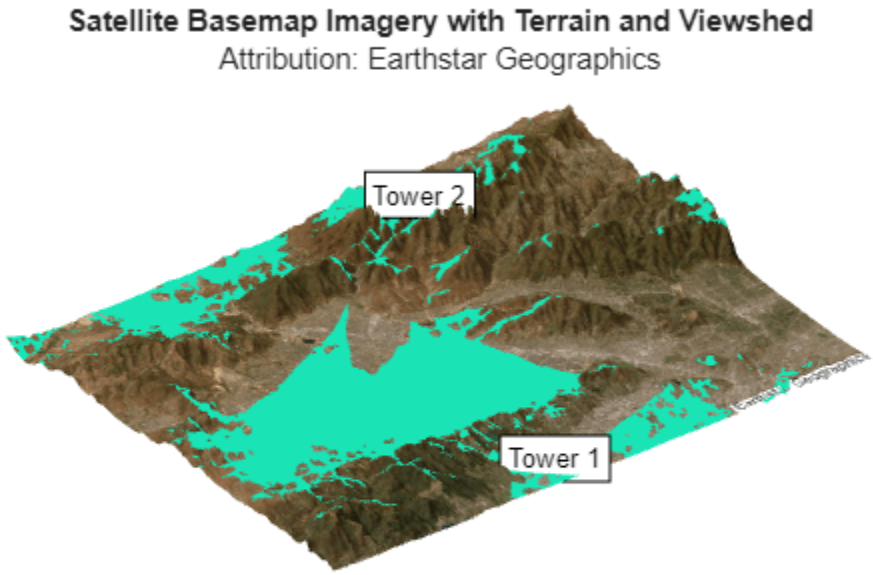
Create a new figure that uses the same colormap as the tiled layout. Remove the axes lines and enable the plot to extend to the edges of the axes.

```
f5 = figure;
f5.Position(3:4) = fSize;

colormap(covcolor)

hold on
axis off
axis tight
```

Visualize the basemap image, viewshed, and terrain using surfaces.

- Display the terrain by draping the basemap image over the elevation data. Specify the z-coordinates using the terrain data, and specify the surface colors using the image data.
- Display the combined viewshed.
- Customize the plot by adding text labels and a title.
- View the plot in 3-D. Adjust the data aspect ratio so that elevation scale is comparable to the *xy*-scale.

```
mapshow(xz,yz,Z,DisplayType="surface",CData=A)

mapshow(xz,yz,vis,EdgeColor="none",DisplayType="surface")

text(x1,y1,z1,text1,Color="k",EdgeColor="k",BackgroundColor="w")
text(x2,y2,z2,text2,Color="k",EdgeColor="k",BackgroundColor="w")
title("Satellite Basemap Imagery with Terrain and Viewshed")
subtitle("Attribution: " + attrib)

daspect([1 1 0.25])
view(3)
```



Visualize Coverage Maps in 2-D

A viewshed is comparable to a wireless coverage map that includes only line-of-sight (LOS) propagation. You can create more detailed coverage maps and calculate received power by using transmitter antenna sites and propagation models.

Calculate Coverage Map

Create a transmitter site for each cell tower. Then, calculate the coverage map and return the result as a `propagationData` object. By default, the `coverage` function uses a Longley-Rice propagation model and incorporates diffraction over terrain.

```
txTower1 = txsite(Name=text1,Latitude=lat1,Longitude=lon1);
txTower2 = txsite(Name=text2,Latitude=lat2,Longitude=lon2);
txsites = [txTower1; txTower2];
pd = coverage(txsites);
```

Extract latitude and longitude values, power values, and power levels from the `propagationData` object by using the `propagationDataGrid` on page 11-33 helper function.

```
dataVariableName = "Power";
maxrange = [30000 30000];
[lats,lons,data,levels] = propagationDataGrid(pd,dataVariableName,maxrange,txsites);
```

Find the limits of the quadrangle that bounds the coverage map.

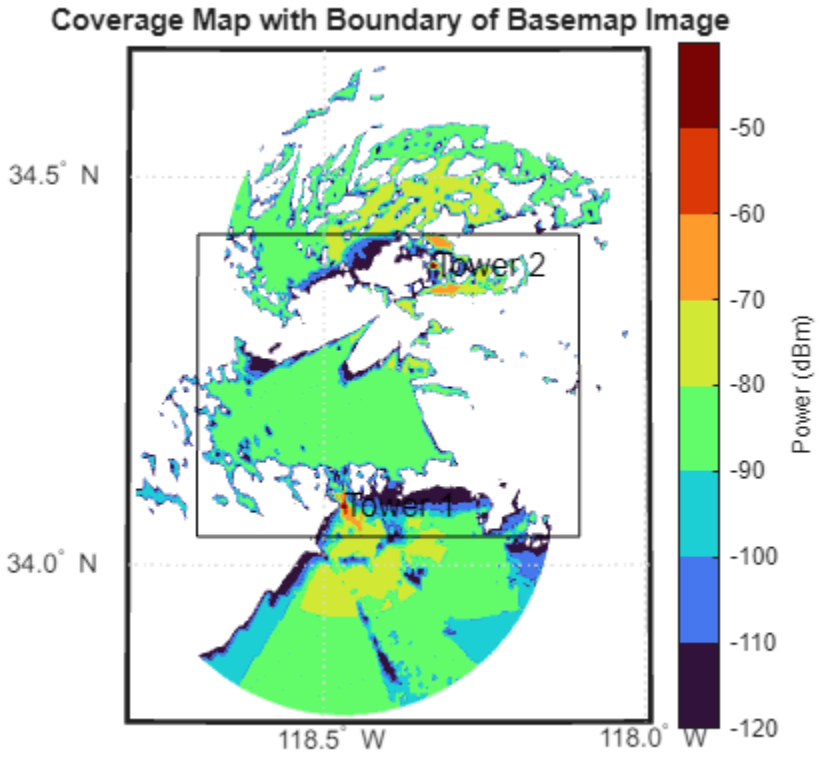
```
covaoi = aoiquad(lats,lons);
[covlatlim,covlonlim] = bounds(covaoi);
```

Define a colormap using the power levels. Create a new figure that uses the power levels colormap.

```
cmap = turbo(length(levels));
climLevels = [min(levels) max(levels)+10];

f6 = figure;
f6.Position(3:4) = fSize;

colormap(cmap)
clim(climLevels)
hold on
```

Create an `axesm`-based map with an appropriate projection for the region. Display the coverage map, a rectangle that represents the San Fernando Valley region, and the transmitter sites on the map. Note that the coverage map extends beyond the San Fernando Valley region.

```
usamap(covlatlim,covlonlim)
geoshow(lats,lons,data,DisplayType="surface")
geoshow(latlim([1 2 2 1 1]),lonlim([1,1,2,2,1]),Color="k")

textm(lat1,lon1,text1,Color="k",FontSize=12)
textm(lat2,lon2,text2,Color="k",FontSize=12)
title("Coverage Map with Boundary of Basemap Image")

cb = colorbar;
cb.Label.String = "Power (dBm)";
cb.Ticks = levels;
```



View Contoured Coverage Map on Basemap Image

Project the geographic coordinates of the coverage map to Web Mercator coordinates.

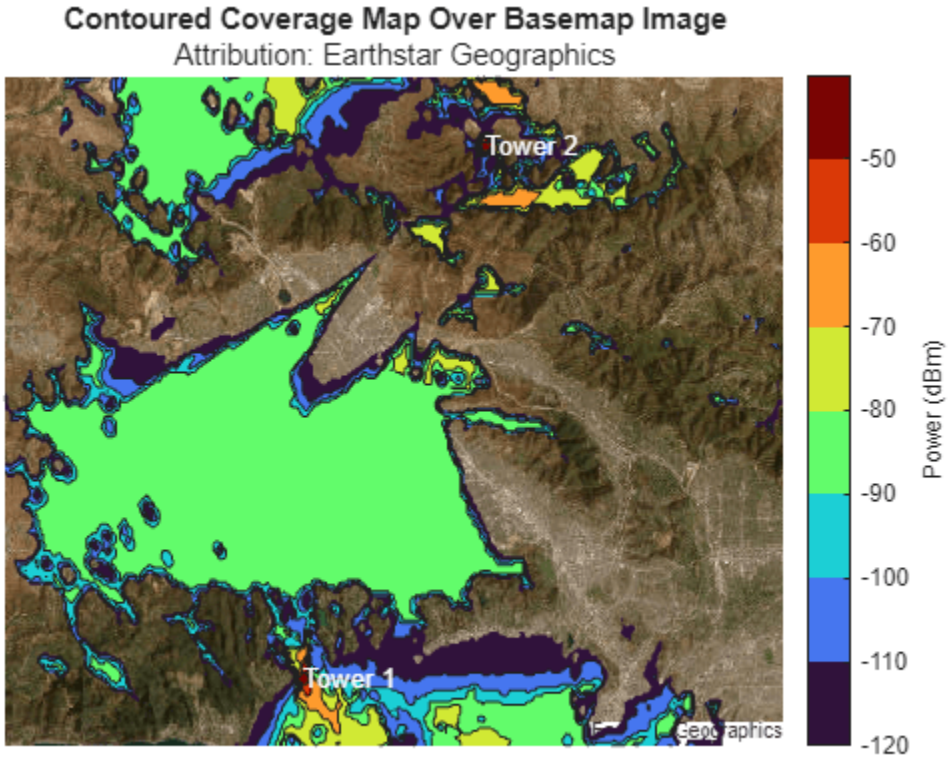
```
[cx,cy] = projfwd(proj,lats,lons);
```

Create a figure that uses the power levels colormap.

```
f7 = figure;
f7.Position(3:4) = fSize;

colormap(cmap)
clim(climLevels)
axis off
hold on
```

Contour the coverage map, specifying the contour levels as the power levels, and display the result over the basemap image. Change the limits of the map to match the San Fernando Valley region.

```
mapshow(A,RA)
contourf(cx,cy,data,LevelList=levels)

xlim(RA.XWorldLimits)
ylim(RA.YWorldLimits)

text(x1,y1,text1,Color="w")
text(x2,y2,text2,Color="w")
title("Contoured Coverage Map Over Basemap Image")
subtitle("Attribution: " + attrib)

cb = colorbar;
cb.Label.String = "Power (dBm)";
cb.Ticks = levels;
```



View Contoured Coverage Map and Viewshed on Basemap Image

Project the geographic coordinates of the combined viewshed to Web Mercator coordinates.

```
[vlat,vlon] = geographicGrid(visR);
[vx,vy] = projfwd(proj,vlat,vlon);
```

Display the coverage map and viewshed over the basemap image by using contour plots. You can create a map with two colorbars by overlaying two axes objects.

Use the first axes to display the basemap image, the coverage map, and a colorbar for the coverage map.

```
f8 = figure;
f8.Position(3:4) = fSize;

ax1 = axes;
hold(ax1,"on")
axis(ax1,"off")
clim(ax1,climLevels)
colormap(ax1,cmap)

mapshow(ax1,A,RA)
contourf(ax1,cx,cy,data,LevelList=levels);

xlim(ax1,RA.XWorldLimits)
ylim(ax1,RA.YWorldLimits)
```

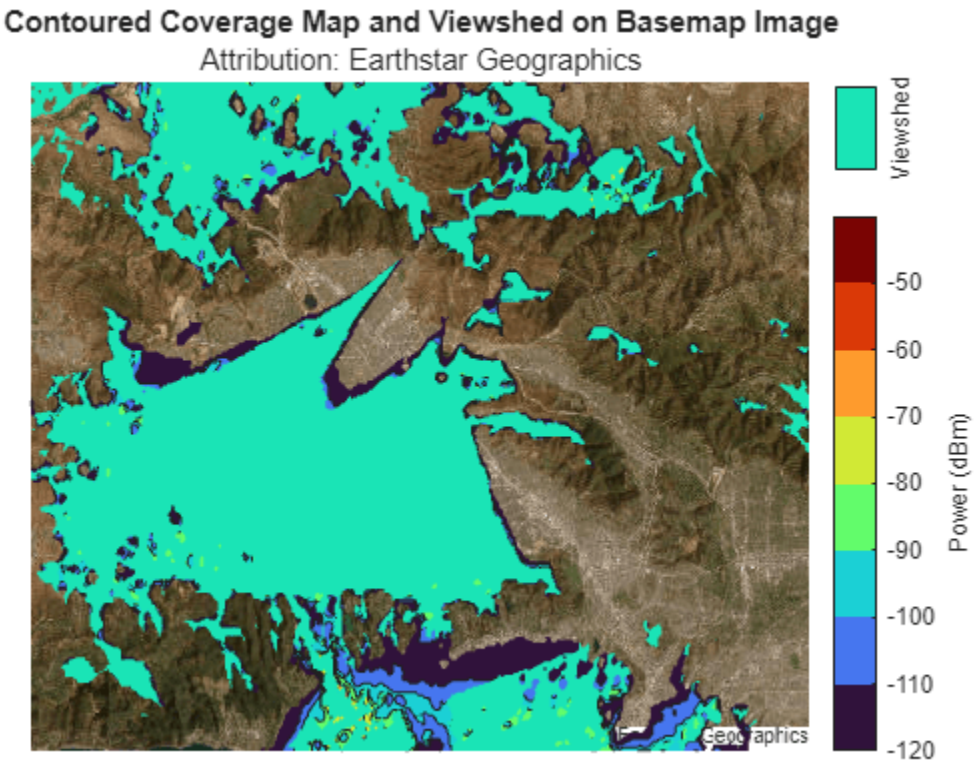
```
title(ax1,"Contoured Coverage Map and Viewshed on Basemap Image")
subtitle("Attribution: " + attrib)

cb1 = colorbar(ax1);
cb1.Label.String = "Power (dBm)";
cb1.Ticks = levels;
```

Use the second axes to display the viewshed and a colorbar for the viewshed.

```
ax2 = axes(Visible="off");
hold(ax2,"on")
colormap(ax2,covcolor)

[~,visContourMap] = contourf(ax2,vx,vy,vis,LevelList=0.5:1:1.5);

cb2 = colorbar(ax2);
cb2.Label.String = "Viewshed";
cb2.Ticks = [];
```

Reposition the axes objects and the colorbars.

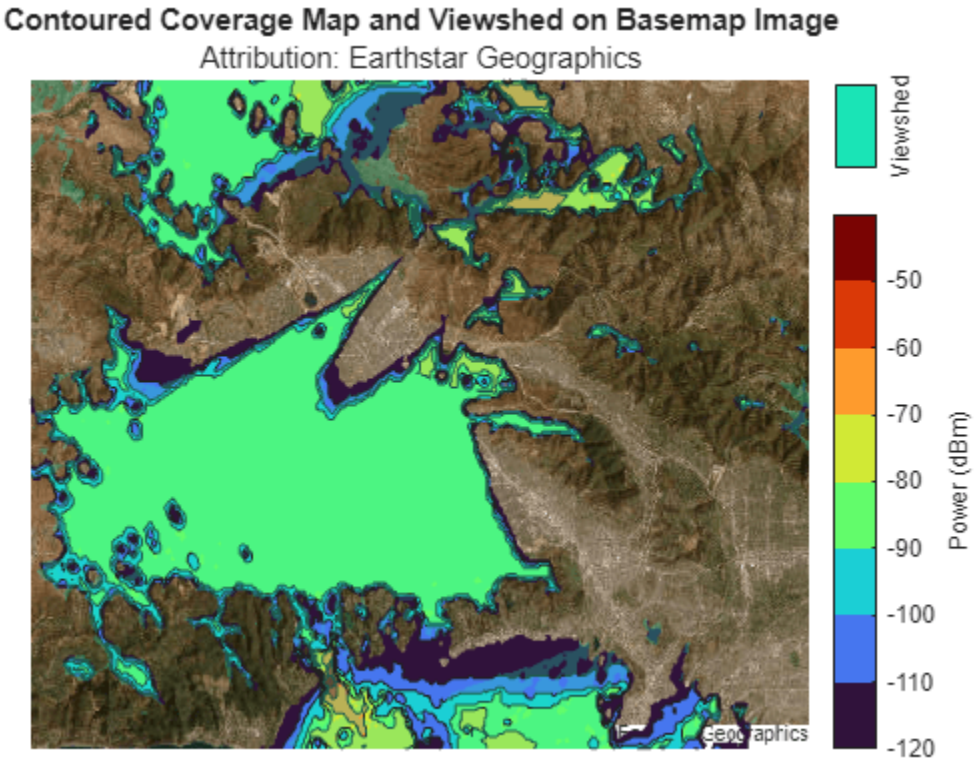
```
ax1.Position = [ax1.Position(1) ax1.Position(2)-0.04 ax1.Position(3) ax1.Position(4)];
ax2.Position = ax1.Position;
cb1.Position = [cb1.Position(1) cb1.Position(2) cb1.Position(3) cb1.Position(4)*0.8];
cb2.Position = [cb2.Position(1) 0.76 cb2.Position(3) 0.1];
```



In addition to LOS analysis, the coverage map incorporates diffraction over terrain. As a result, the coverage map extends beyond the viewshed.

Adjust the transparency of the viewshed. When you decrease the value of `faceAlpha`, the viewshed becomes more transparent and you can see the coverage map beneath.

```
faceAlpha = 0.3 ;
visContourMap.FaceAlpha = faceAlpha;
```



View Contoured Coverage Map on Geographic Axes

To display a contoured coverage map over geographic axes, contour the coverage map using the `contourDataGrid` on page 11-34 helper function. Each row of the returned geospatial table corresponds to a power level and includes the contour shape, the area of the contour shape in square kilometers, the minimum power for the level, and the power range for the level. The area value for each contour shape includes the regions that are covered by other contour shapes.

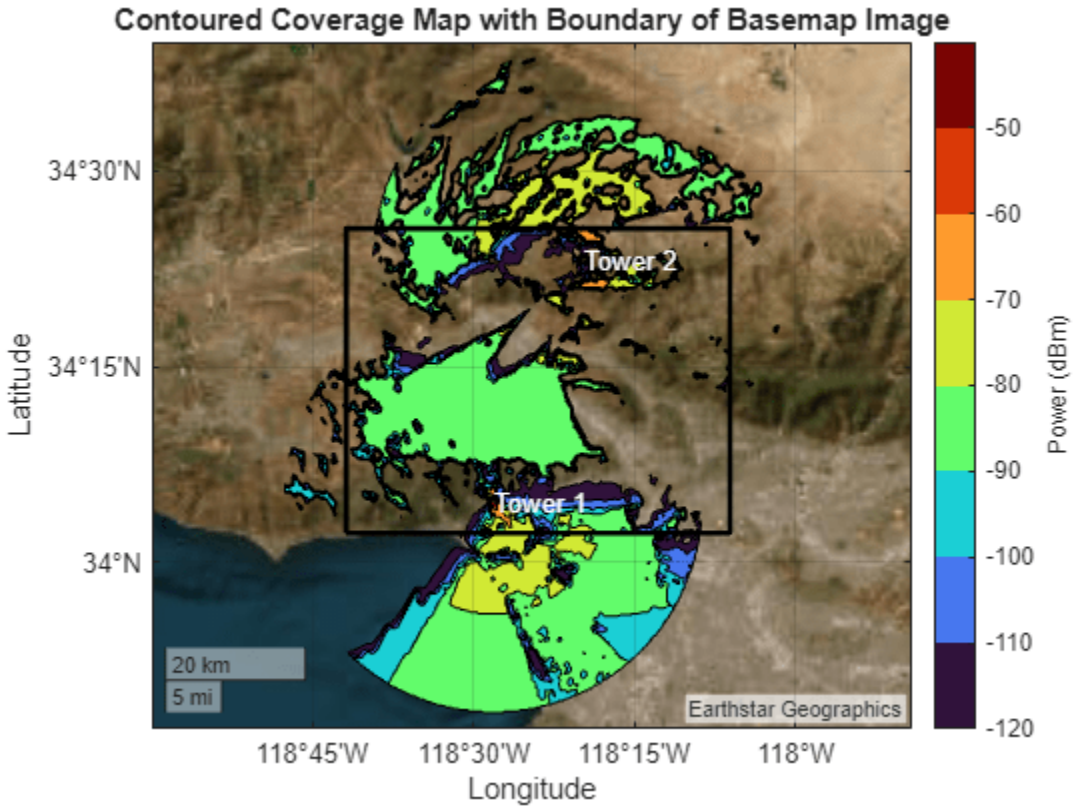
```
GT = contourDataGrid(lats,lons,data,levels);
disp(GT)
```

Shape	Area (sq km)	Perimeter (km)	Power (dBm)	Power Range (dBm)	
geopolyshape	2588.9	1.5139e+06	-120	-120	-110
geopolyshape	2314	1.5045e+06	-110	-110	-100
geopolyshape	2134.4	1.5159e+06	-100	-100	-90
geopolyshape	1712.9	1.4211e+06	-90	-90	-80
geopolyshape	309.79	5.0189e+05	-80	-80	-70
geopolyshape	13.563	45401	-70	-70	-60

```
    geopolyshape       1.8949            9214.1          -60            -60      -50
    geopolyshape      0.67629            4408.2          -50            -50      -40
```

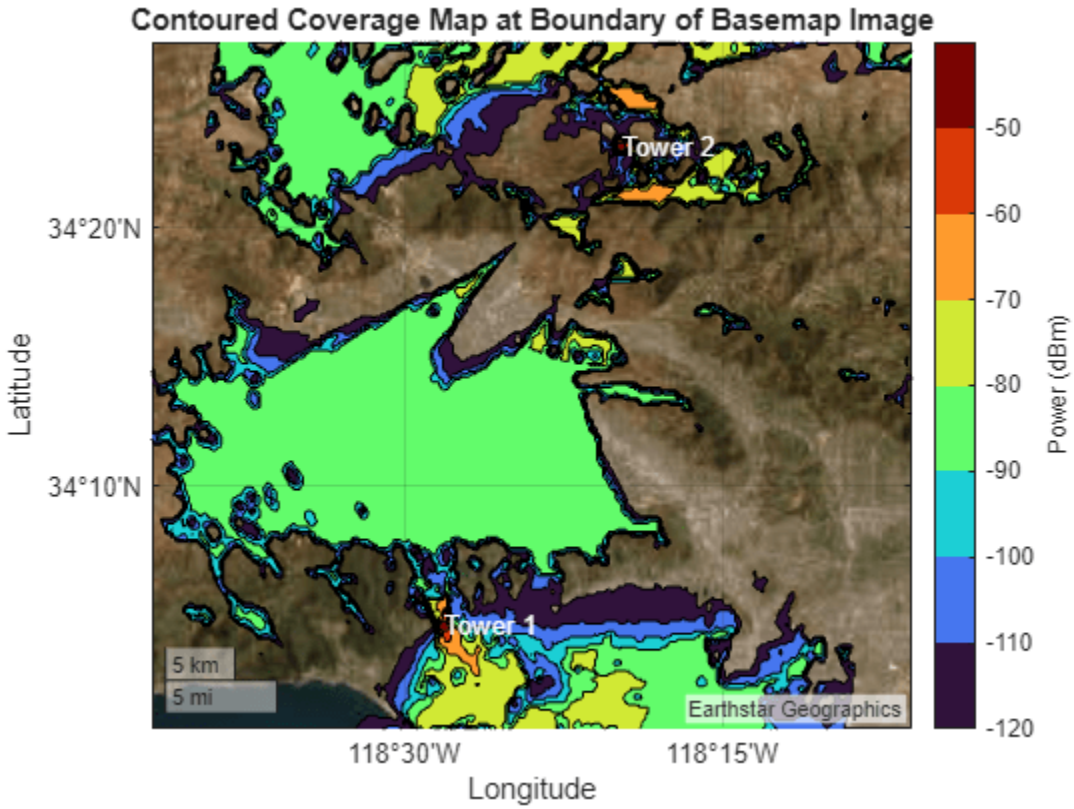
Plot the contoured coverage map and a rectangle that represents the San Fernando Valley region into geographic axes.

```
f9 = figure;
f9.Position(3:4) = fSize;

geobasemap satellite
colormap(cmap)
clim(climLevels)
hold on

hp = geoplot(GT,ColorVariable="Power (dBm)",EdgeColor="k",FaceAlpha=1);
boundary = geoplot(latlim([1 2 2 1 1]),lonlim([1,1,2,2,1]),Color="k",LineWidth=2);

text(lat1,lon1,text1,Color="w")
text(lat2,lon2,text2,Color="w")
title("Contoured Coverage Map with Boundary of Basemap Image")

cb = colorbar;
cb.Label.String = "Power (dBm)";
cb.Ticks = levels;
```



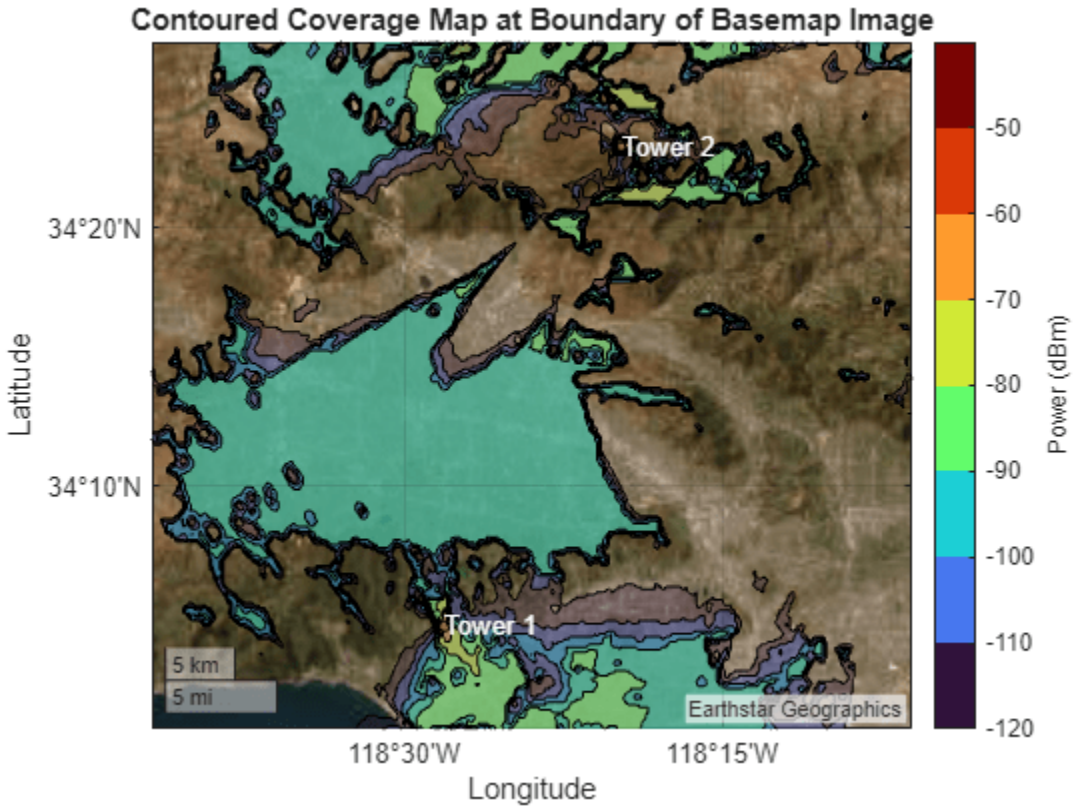
Zoom the map into the San Fernando Valley region and delete the boundary.

```
geolimits(latlim,lonlim)
delete(boundary)
title("Contoured Coverage Map at Boundary of Basemap Image")
```



Adjust the transparency of the coverage map. When you decrease the value of `faceAlpha`, the coverage map becomes more transparent and you can see the basemap beneath.

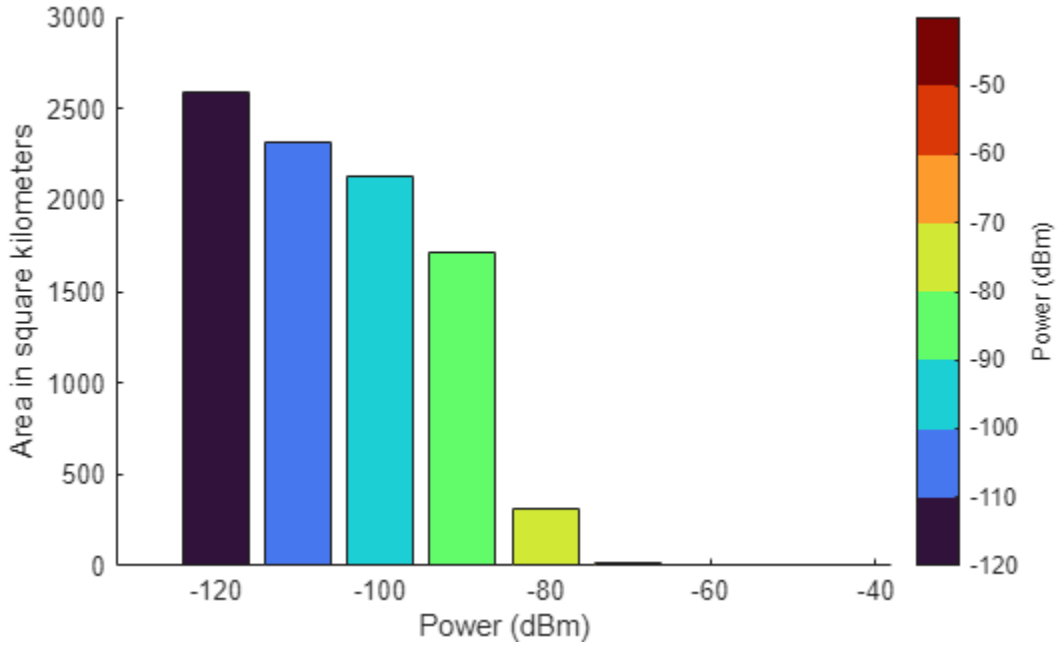
```
faceAlpha = 0.3 ;
hp.FaceAlpha = faceAlpha;
```



View Bar Graph of Coverage Area

Create a bar graph that shows the coverage area in square kilometers for each power level. As the power level increases, the coverage area decreases.

```
figure
colormap(cmap)
clim(climLevels)
hold on

bar(GT.("Power (dBm)"),GT.("Area (sq km)"),FaceColor="flat",CData=cmap);
ylabel("Area in square kilometers")
xlabel("Power (dBm)")

cb = colorbar;
cb.Label.String = "Power (dBm)";
cb.Ticks = levels;
```



Determine If Points Are Within Coverage

Determine whether points within the San Fernando Valley region are within coverage.

Calculate Coverage at Point Locations

Specify the locations of two points near the cell towers. Determine whether the points are within coverage by using the `incoverage` on page 11-37 helper function. The result indicates the first point is in coverage and the second point is not in coverage.

```
locationLats = [34.2107 34.1493];
locationLons = [-118.4545 -118.1744];

[tf,powerLevels] = incoverage(locationLats,locationLons,GT);
disp(tf)
```

```
   1
   0
```

Display the power levels. The result `-Inf` indicates that the second point is not in coverage.

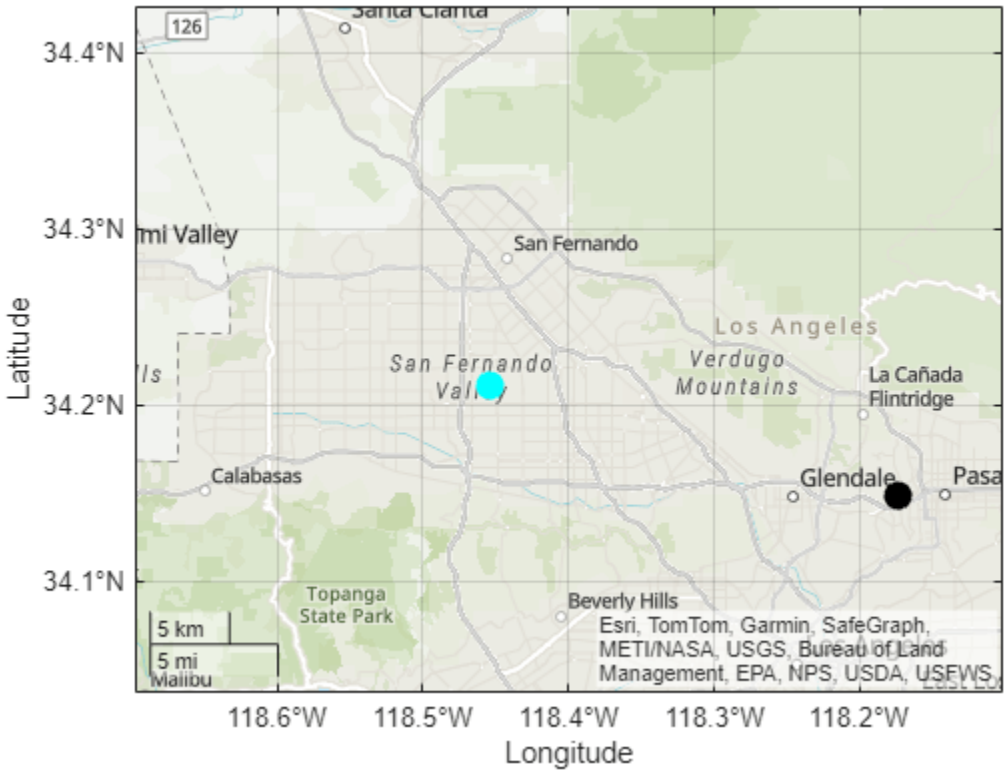
```
disp(powerLevels)
```

```
   -90
  -Inf
```

Display the point that is in coverage using a cyan marker and the point that is not in coverage using a black marker.

```
f10 = figure;
f10.Position(3:4) = fSize;

geobasemap topographic
hold on
```

```
geotickformat dd
geolimits(latlim,lonlim)

geoplot(locationLats(tf),locationLons(tf),"o", ...
    MarkerEdgeColor="c",MarkerFaceColor="c",MarkerSize=10)
geoplot(locationLats(~tf),locationLons(~tf),"o", ...
    MarkerEdgeColor="k",MarkerFaceColor="k",MarkerSize=10)
```



Interactively Select Points and Calculate Coverage at Point Locations

Select points and determine whether they are in coverage.

Set up a map that uses the power levels colormap. Then, plot the contoured coverage map.

```
f11 = figure;
f11.Position(3:4) = fSize;

geobasemap topographic
geolimits(latlim,lonlim)
hold on
geotickformat dd
colormap(cmap)
clim(climLevels)

contourPlot = geoplot(GT,ColorVariable="Power (dBm)",EdgeColor="k",FaceAlpha=1);

cb = colorbar;
```

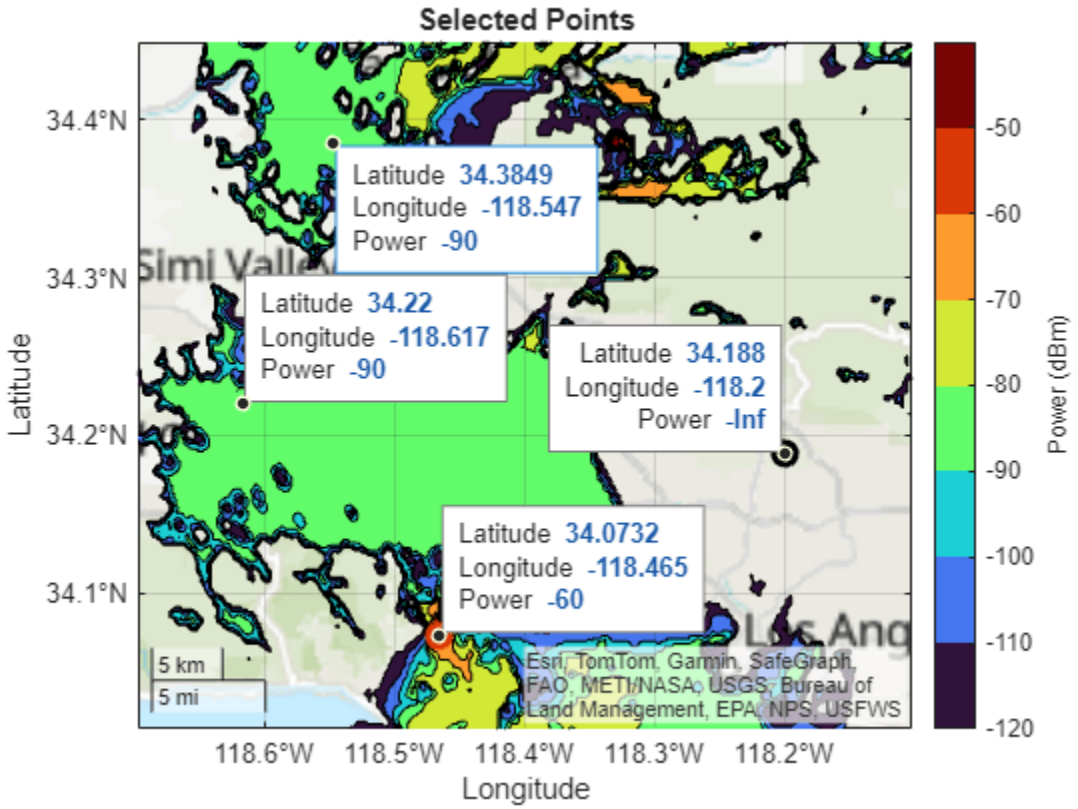
```
cb.Label.String = "Power (dBm)";
cb.Ticks = levels(1:end);
```

Select the points. Use predefined points by specifying `interactivelySelectPoints` as `false`. Alternatively, you can interactively select points by specifying `interactivelySelectPoints` as `true`. Specify the number of points to interactively select by specifying a value for `numberOfPoints`.

```
interactivelySelectPoints = false;
if interactivelySelectPoints
    numberOfPoints = 4; %#ok<*UNRCH>
    title("Select " + string(numberOfPoints) + " Points")
    pts = ginput(numberOfPoints);
else
    pts = [ ...
        34.0732 -118.4652
        34.1880 -118.2000
        34.2200 -118.6172
        34.3849 -118.5472];
end
```

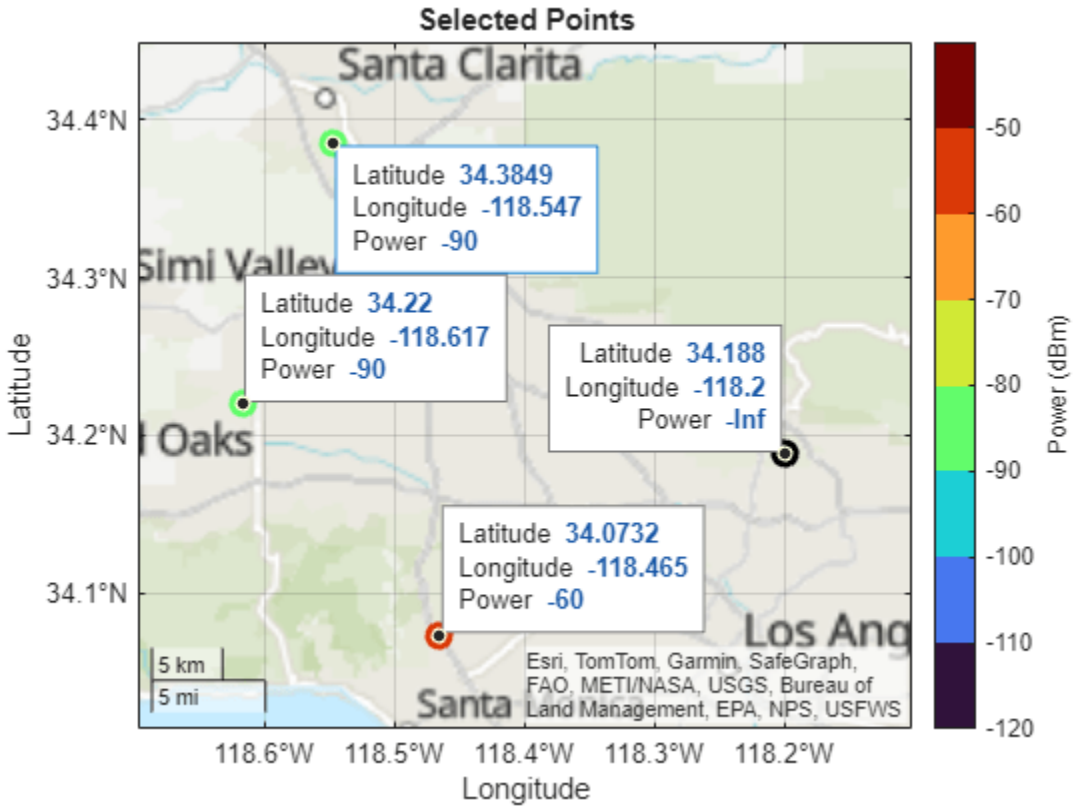
Determine whether each point is within coverage. If the point is not within coverage, display it on the map in black. If the point is within coverage, display it on the map using the color associated with its power level.

```
for k = 1:size(pts,1)
    latpoint = pts(k,1);
    lonpoint = pts(k,2);
    [tf,powerLevel] = incoverage(latpoint,lonpoint,GT);

    if ~tf
        color = [0 0 0];
    else
        index = GT.("Power (dBm)") == powerLevel;
        color = cmap(index,:,:);
    end

    h = geoplot(latpoint,lonpoint,"o",MarkerEdgeColor=color, ...
        MarkerFaceColor=color,MarkerSize=10);
    dt = datatip(h);
    dtrow = dataTipTextRow("Power",powerLevel);
    h.DataTipTemplate.DataTipRows(end+1) = dtrow;
end
title("Selected Points")
```



Toggle off the contour plot by setting `showContourPlot` to `false`. The color of each point indicates its power level.

```
showContourPlot = false;
contourPlot.Visible = showContourPlot;
```



Create Geographic Coverage Maps for Other Regions

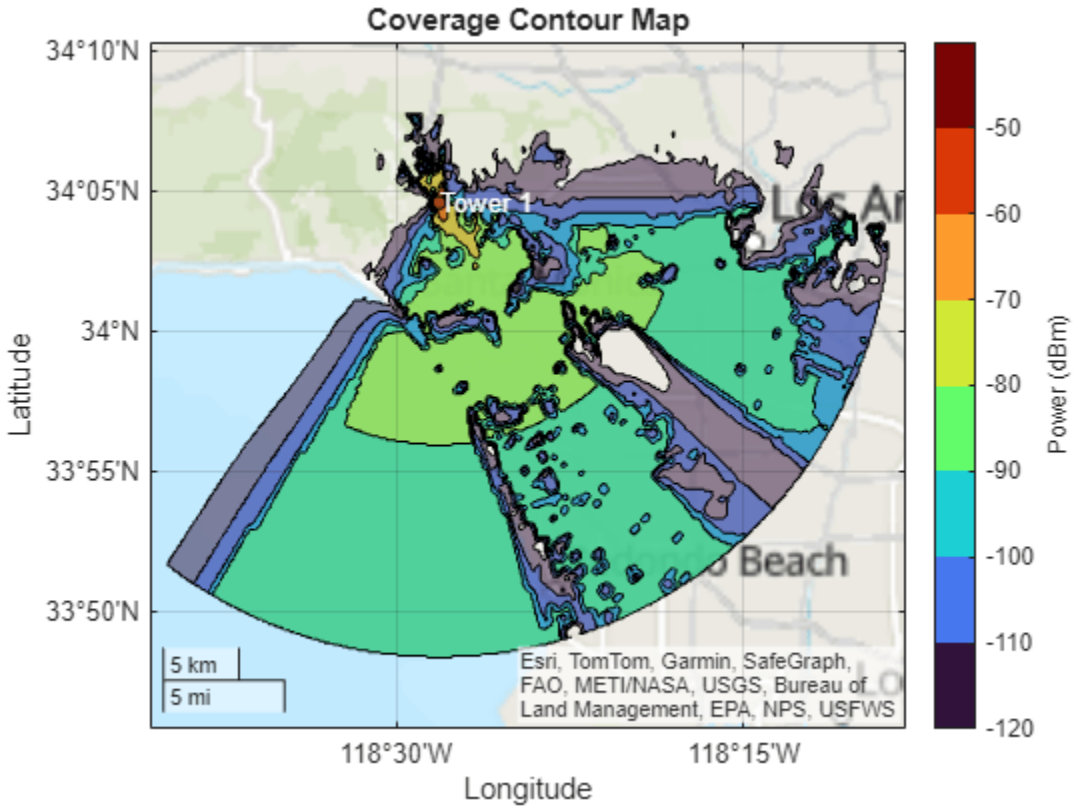
The `geocoverage` on page 11-37 helper function calculates or displays a coverage map for the specified transmitter sites. When you specify an output argument, the function returns a geospatial table.

```
T = geocoverage(txsites);
disp(T)
```

Shape	Area (sq km)	Perimeter (km)	Power (dBm)	Power Range (dBm)	
geopolyshape	2588.9	1.5139e+06	-120	-120	-110
geopolyshape	2314	1.5045e+06	-110	-110	-100
geopolyshape	2134.4	1.5159e+06	-100	-100	-90
geopolyshape	1712.9	1.4211e+06	-90	-90	-80
geopolyshape	309.79	5.0189e+05	-80	-80	-70
geopolyshape	13.563	45401	-70	-70	-60
geopolyshape	1.8949	9214.1	-60	-60	-50
geopolyshape	0.67629	4408.2	-50	-50	-40

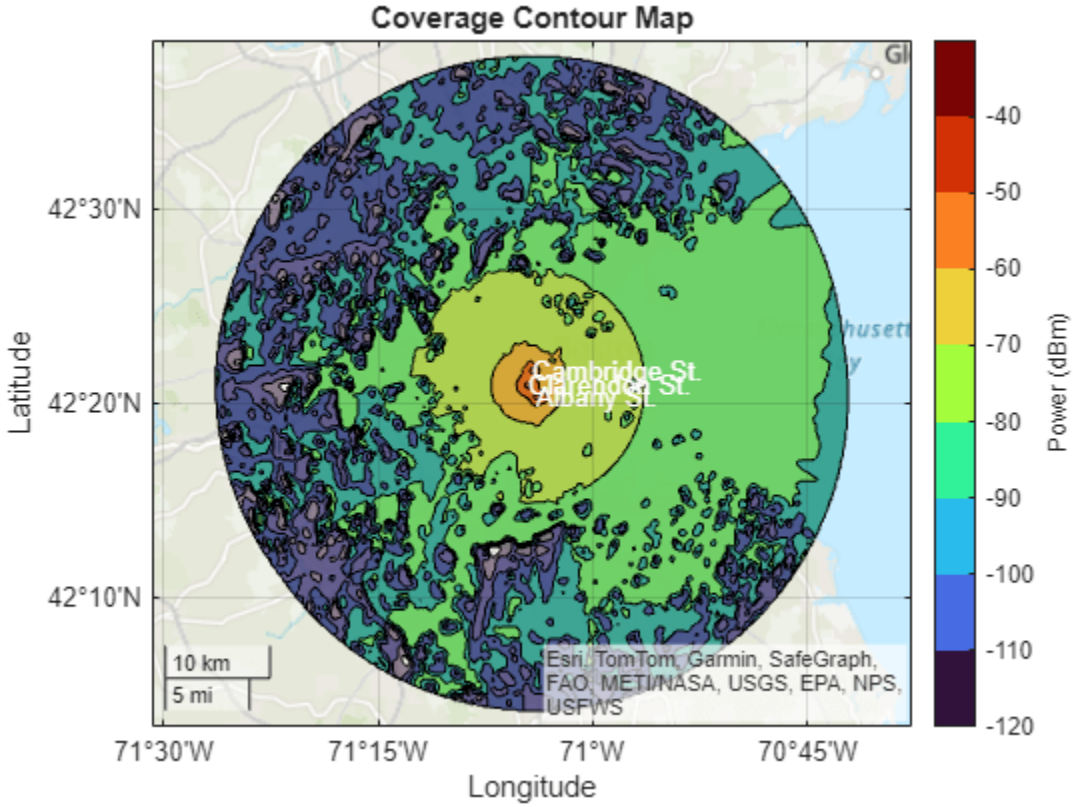
When you omit the output argument, the function contours the coverage map and displays the result over a basemap.

```
basemap = "topographic";
geocoverage(txsites(1),basemap)
```



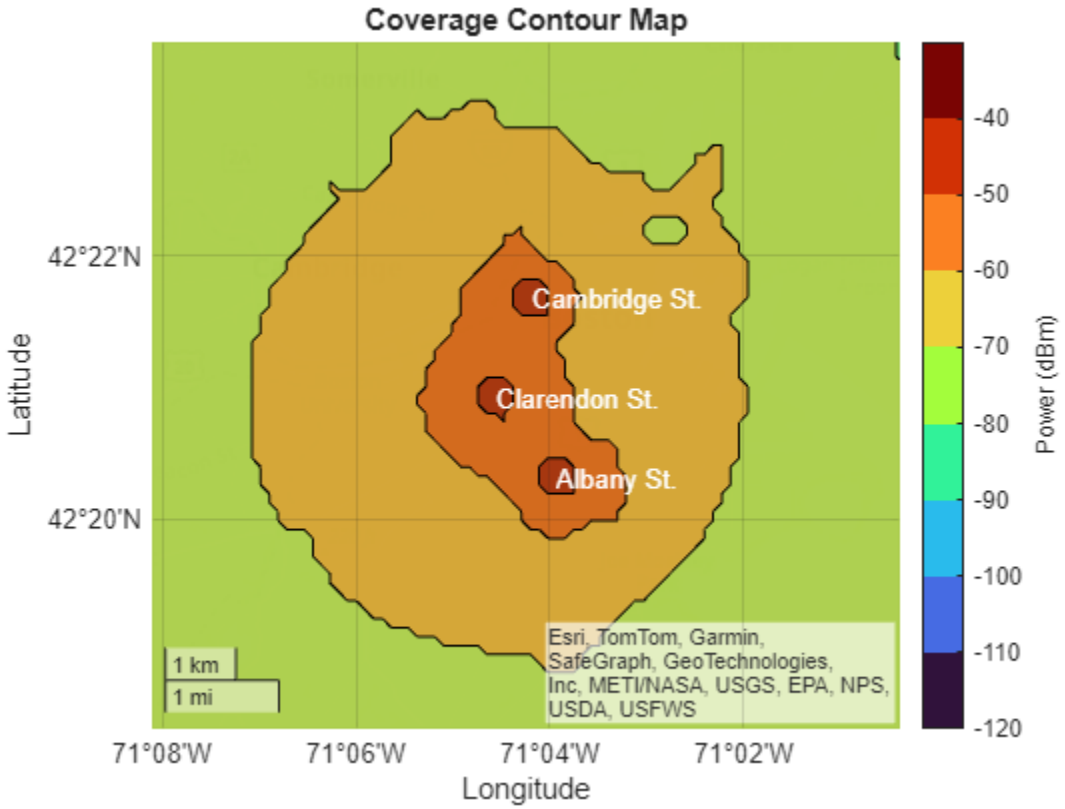
You can use the `geocoverage` helper function to create coverage maps for other locations. For example, create a coverage map for cell towers near Boston. The `geocoverage` function calculates coverage data for up to 30 kilometers from the transmitter sites.

```
names = ["Clarendon St.","Cambridge St.","Albany St."];
bostonLat = [42.348722 42.361222 42.338444];
bostonLon = [-71.075889 -71.069778 -71.065611];
bostonH = [260 30 23];
freq = [852.637e6 862.012e6 862.012e6];
txs = txsite(Name=names,Latitude=bostonLat,Longitude=bostonLon, ...
             AntennaHeight=bostonH,TransmitterFrequency=freq);
geocoverage(txs,basemap)
```



The cell towers are close together. View the area immediately around the towers by zooming in on the coverage map.

```
gx = gca;
gx.ZoomLevel = 12;
```



Helper Functions

The `propagationDataGrid` helper function extracts the latitude and longitude values `clats` and `clons`, the power values `cdata`, and the power levels `levels` from the propagation data object `pd` with data variable `dataVariableName`, using the transmitter sites `txsites` and the maximum range in meters from the transmitter sites `maxrange`. Within the function:

- Create regularly spaced grids of latitude and longitude values from the propagation data object.
- Determine which latitude and longitude values are within range of the transmitter sites.
- Create a grid of power values by interpolating the propagation data. Interpolate only the values that are within range of the transmitter sites.
- Determine the power levels by discretizing the data grid.

```
function [clats,clons,cdata,levels] = propagationDataGrid(pd,dataVariableName,maxrange,txsites)
% Extract propagation data grid from propagationData object

    % Create grids of latitude and longitude values
    imageSize = [512 512];
    lats = pd.Data.Latitude;
    lons = pd.Data.Longitude;
    data = pd.Data.(dataVariableName);

    [lonmin,lonmax] = bounds(lons);
    [latmin,latmax] = bounds(lats);
```

```
    lonsv = linspace(lonmin,lonmax,imageSize(2));
    latsv = linspace(latmin,latmax,imageSize(1));
    [clons,clats] = meshgrid(lonsv,latsv);

    % Determine which latitude and longitude values are within range of the
    % transmitters
    txlat = [txsites.Latitude]';
    txlon = [txsites.Longitude]';

    numTx = size(txlat,1);
    gc = nan(numel(clons),numTx);
    for txInd = 1:numTx
        gc(:,txInd) = distance(txlat(txInd),txlon(txInd),clats(:),clons(:),wgs84Ellipsoid);
    end
    isInRange = any(gc <= maxrange,2);

    % Create grid of power values
    cdata = nan(imageSize);
    cdata(isInRange) = interp(pd,clats(isInRange),clons(isInRange), ...
        DataVariableName=dataVariableName);

    % Determine power levels
    [bmin,bmax] = bounds(data);
    levels = max(bmin,-120):10:bmax;
    datalevels = sort(levels);
    % Need max bin edge to include all values
    maxBin = max(max(datalevels(:)),max(cdata(:))) + 1;
    bins = [datalevels(:); maxBin];
    cdata = discretize(cdata,bins,datalevels);
end
```

The `contourDataGrid` helper function contours a data grid created using the `propagationDataGrid` helper function and returns the result as a geospatial table. Each row of the table corresponds to a power level and includes the contour shape, the area of the contour shape in square kilometers, the minimum power for the level, and the power range for the level. Within the function:

- Ensure that the function contours power values below the minimum power level by replacing `NaN` values in the power grid with a large negative number.
- Prepare to contour the data by projecting the latitude and longitude coordinates to *xy*-coordinates. Contouring *xy*-coordinates is more performant than contouring latitude and longitude coordinates.
- Contour the projected coordinates using the `contourf` function. Specify the contour levels as the power levels. The `contourf` function returns a matrix that contains, for each power level, the coordinates of the contour lines.
- Prepare to process the data for each contour line by initializing several variables.
- Process the data for each contour line. Remove contour lines made of collinear points or fewer than three points. Unproject the coordinates.
- Create contour shapes from the unprojected coordinates. Include the contour shapes, the areas of the shapes, the perimeters of the shapes, and the power levels in a geospatial table.
- Condense the geospatial table into a new geospatial table with one row per power level. Each row contains one combined contour shape (that includes all contour shapes for the power level), the

total area of the combined contour shape, the total perimeter of the combined contour shape, the minimum power for the level, and the power range for the level.

- Clean up the geospatial table. Convert the area of each shape to square kilometers, ensure that small contours appear on top of large contours by sorting the rows, and rename the table variables.

```
function GT = contourDataGrid(latd,lond,data,levels)
% Contour data grid created from propagationDataGrid helper function

    % Replace NaN values in power grid with a large negative number
    data(isnan(data)) = min(levels) - 1000;

    % Project the coordinates using an equal-area projection
    proj = projcrs(102008,"Authority","ESRI");
    [xd,yd] = projfwd(proj,latd,lond);

    % Contour the projected data
    fig = figure("Visible","off");
    obj = onCleanup(@()close(fig));
    c = contourf(xd,yd,data,LevelList=levels);

    % Initialize variables
    x = c(1,:);
    y = c(2,:);
    n = length(y);
    k = 1;
    index = 1;
    levels = zeros(n,1);
    latc = cell(n,1);
    lonc = cell(n,1);

    % Process the coordinates of the projected contours
    while k < n
        level = x(k);
        numVertices = y(k);
        yk = y(k+1:k+numVertices);
        xk = x(k+1:k+numVertices);
        k = k + numVertices + 1;

        [first,last] = findNanDelimitedParts(xk);
        jindex = 0;
        jy = {};
        jx = {};

        for j = 1:numel(first)
            % Process the j-th part of the k-th level
            s = first(j);
            e = last(j);
            cx = xk(s:e);
            cy = yk(s:e);
            if cx(end) ~= cx(1) && cy(end) ~= cy(1)
                cy(end+1) = cy(1); %#ok<*AGROW>
                cx(end+1) = cx(1);
            end

            if j == 1 && ~ispolycw(cx,cy)
                % First region must always be clockwise
```

```
            [cx,cy] = poly2cw(cx,cy);
        end

        % Filter regions made of collinear points or fewer than 3 points
        if ~(length(cx) < 4 || all(diff(cx) == 0) || all(diff(cy) == 0))
            jindex = jindex + 1;
            jy{jindex,1} = cy(:)';
            jx{jindex,1} = cx(:)';
        end

    end

    % Unproject the coordinates of the processed contours
    [jx,jy] = polyjoin(jx,jy);
    if length(jy) > 2 && length(jx) > 2
        [jlat,jlon] = projinv(proj,jx,jy);
        latc{index,1} = jlat(:)';
        lonc{index,1} = jlon(:)';
        levels(index,1) = level;
        index = index + 1;
    end
end

% Create contour shapes from the unprojected coordinates. Include the
% contour shapes, the areas of the shapes, the perimeters of the
% shapes, and the power levels in a geospatial table.
latc = latc(1:index-1);
lonc = lonc(1:index-1);
Shape = geopolyshape(latc,lonc);

Area = area(Shape);

Perimeter = perimeter(Shape);

levels = levels(1:index-1);
Levels = levels;

allPartsGT = table(Shape,Area,Perimeter,Levels);

% Condense the geospatial table into a new geospatial table with one
% row per power level.
GT = table.empty;
levels = unique(allPartsGT.Levels);
for k = 1:length(levels)
    gtLevel = allPartsGT(allPartsGT.Levels == levels(k),:);
    tLevel = geotable2table(gtLevel,["Latitude","Longitude"]);
    [lon,lat] = polyjoin(tLevel.Longitude,tLevel.Latitude);
    Shape = geopolyshape(lat,lon);
    Levels = levels(k);
    Area = area(Shape);
    Perimeter = perimeter(Shape);
    GT(end+1,:) = table(Shape,Area,Perimeter,Levels);
end

maxLevelDiff = max(abs(diff(GT.Levels)));
LevelRange = [GT.Levels GT.Levels + maxLevelDiff];
GT.LevelRange = LevelRange;
```

```
    % Clean up the geospatial table
    GT.Area = GT.Area/1e6;
    GT = sortrows(GT,"Area","descend");
    GT.Properties.VariableNames = ...
        ["Shape","Area (sq km)","Perimeter (km)","Power (dBm)","Power Range (dBm)"];
end
```

The `incoverage` helper function determines whether the points with latitude values `lat` and longitude values `lon` are within the coverage map defined by the geospatial table `T`. Within the function:

- Determine whether the points are within the coverage map and return the results in `in`. The elements of `in` are `1` (`true`) when the corresponding points are within coverage.
- Determine the power levels for the points and return the results in `powerLevels`. A value of `-Inf` indicates that the corresponding point is not within coverage.

```
function [in,powerLevels] = incoverage(lat,lon,T)
% Query points in coverage

    % Determine whether points are within coverage
    tf = false(length(T.Shape),length(lat));
    for k = 1:length(T.Shape)
        tf(k,:) = isinterior(T.Shape(k),geopointshape(lat,lon));
    end

    % Determine the power levels for the points
    in = false(length(lat),1);
    powerLevels = -inf(length(lat),1);
    for k = 1:length(in)
        v = tf(:,k);
        in(k) = any(v);
        if in(k)
            powerLevels(k) = max(T{v,"Power (dBm)"});
        end
    end
end
```

The `geocoverage` helper function calculates or displays a coverage map for the transmitter sites `txsites`. Within the function:

- Calculate the coverage map by using the `coverage` function. Then, contour the coverage map by using the `progagationDataGrid` and `contourDataGrid` helper functions.
- When you do not specify an output argument, the function displays the contoured coverage map over geographic axes using the basemap `basemap`.
- When you do specify an output argument, the function returns the result as a geospatial table.

```
function varargout = geocoverage(txsites,basemap)
% Display or calculate contoured coverage map

    % Calculate and contour the coverage map
    pd = coverage(txsites);
    dataVariableName = "Power";
```

```
    maxrange = 30000*ones(1,length(txsites));
    [lats,lons,data,levels] = propagationDataGrid(pd,dataVariableName,maxrange,txsites);
    GT = contourDataGrid(lats,lons,data,levels);

    % If you do not specify an output argument, display the coverage map
    if nargout == 0
        if nargin < 2
            basemap = "satellite";
        end

        cmap = turbo(length(levels));
        fig = figure;
        fig.Position(3:4) = [560 420];

        geobasemap(basemap)

        geoplot(GT,ColorVariable="Power (dBm)",EdgeColor="k",FaceAlpha=0.5)
        colormap(cmap)
        clim([min(levels) max(levels)+10])

        h = colorbar;
        h.Label.String = "Power (dBm)";
        h.Ticks = levels;

        for k = 1:length(txsites)
            latt = txsites(k).Latitude;
            lont = txsites(k).Longitude;
            text(latt,lont,txsites(k).Name,Color="w")
        end

        title("Coverage Contour Map")
    % If you specify an output argument, return a geospatial table
    else
        varargout{1} = GT;
    end
end
```

The `findNanDelimitedParts` helper function finds the indices of the first and last elements of each `NaN`-separated part of the array `x`. The `contourDataGrid` helper function uses `findNanDelimitedParts`.

```
function [first,last] = findNanDelimitedParts(x)
% Find indices of the first and last elements of each part in x.
% x can contain runs of multiple NaNs, and x can start or end with
% one or more NaNs.

    n = isnan(x(:));

    firstOrPrecededByNaN = [true; n(1:end-1)];
    first = find(~n & firstOrPrecededByNaN);

    lastOrFollowedByNaN = [n(2:end); true];
```

```
    last = find(~n & lastOrFollowedByNaN);
end
```

See Also

Functions
`viewshed` | `contourf` | `coverage` | `geoplot` | `readBasemapImage`

Objects
`txsite` | `LongleyRice`

Related Examples

- “Create Geospatial Tables” on page 6-2
- “Create Common Plots over Basemap Images” on page 4-52

Visualize Coverage Maps over Lunar Terrain Using Ray Tracing

Display viewsheds and coverage maps for a base station on the surface of the Moon using terrain data and ray tracing analysis. A *viewshed* includes areas that are within line of sight (LOS) of a set of points, and a *coverage map* includes received power for the areas within range of a set of wireless transmitters. You can use a viewshed as a first-order approximation of a coverage map.

The goal of this example is to analyze radio coverage for a lunar module during several extra-vehicular activities (EVAs). During these EVAs, a moon rover travels from the lunar module to nearby hill clusters. The trajectories of the moon rover are based on the 1971 Apollo 15 mission, which was the first Apollo mission to use a moon rover. Because the moon rover enabled EVAs of longer distances than previous Apollo missions, maintaining communications between the rover, the module, and Earth was critical to the mission. During the mission, the rover frequently drove through areas with no radio coverage, inhibiting communication with the module.

When using geographic coordinates, the ray tracing capabilities in Antenna Toolbox™ and Communications Toolbox™ use an Earth-based area of interest (AOI). By combining the ray tracing capabilities with the mapping capabilities in Mapping Toolbox™, you can visualize coverage over generic planetary terrain.

This example shows you how to:

- Read lunar terrain data and a reference spheroid from a GeoTIFF file.
- Get a reference ellipsoid for the Moon.
- Generate the trajectory of a moon rover.
- View terrain, coverage maps, and viewsheds in 2-D.
- View terrain and propagation paths in 3-D.

Read Terrain and Reference Spheroid

Read terrain elevation data for the AOI and a reference spheroid for the moon. The reference spheroid enables you to generate the trajectory of the moon rover, to perform the ray tracing analysis, and to create maps of the region.

The Apollo 15 mission took place in a region surrounding the Hadley Plains. Read lunar terrain elevation data [1] on page 11-56 for the Hadley Plains from a GeoTIFF file. The file reports elevations in meters with respect to a reference spheroid that represents the shape of the Moon. You can download the data in JPEG 2000 format from the Goddard Space Flight Center [2] on page 11-56. The file used in this example has been cropped from the original data set and exported to a GeoTIFF file.

```
filename = "SLDEM2015_Hadley_Plains_Apollo15.tif";
[Z,R] = readgeoraster(filename,OutputType="double");
```

The `readgeoraster` function imports the data into the workspace as an array and a raster reference object in geographic coordinates. The reference object stores information such as the latitude limits, the longitude limits, and the geographic coordinate reference system (CRS). Display the properties of the reference object.

```
disp(R)
```

```
  GeographicCellsReference with properties:
```

```
            LatitudeLimits: [24.6328125 27.630859375]
           LongitudeLimits: [2.134765625 5.1328125]
                RasterSize: [307 307]
       RasterInterpretation: 'cells'
           ColumnsStartFrom: 'north'
              RowsStartFrom: 'west'
      CellExtentInLatitude: 0.009765625
     CellExtentInLongitude: 0.009765625
    RasterExtentInLatitude: 2.998046875
   RasterExtentInLongitude: 2.998046875
          XIntrinsicLimits: [0.5 307.5]
          YIntrinsicLimits: [0.5 307.5]
     CoordinateSystemType: 'geographic'
             GeographicCRS: [1×1 geocrs]
                 AngleUnit: 'degree'
```

The geographic CRS stores the reference spheroid for the Moon. Extract the reference spheroid from the CRS.

```
moonSpheroid = R.GeographicCRS.Spheroid;
```

Define Lunar Module, Moon Rover, and Trajectory of Rover

Specify the latitude and longitude coordinates [3] on page 11-56 of the lunar module. Approximate the height of the lunar module in meters.

```
moduleLat = 26.132;
moduleLon = 3.634;
moduleHeight = 7;
```

The lunar module uses the Unified-S band radio system for tracking and communication. Specify the frequency in hertz and the power in watts [4] on page 11-56.

```
moduleFreq = 2.82e9;
modulePower = 20;
```

Approximate the height of the moon rover in meters.

```
roverHeight = 1.5;
```

Approximate the trajectories of three EVAs: one to the Hadley Rille, another to the Mons Hadley Delta hill complex, and the last to the South Cluster hill complex. For each trajectory:

- Specify the approximate coordinates of the rover destination.
- Calculate the trajectory by finding the coordinates of points between the lunar module and the rover destination. Get points along a geodesic on the Moon by using the `track2` function and specifying the Moon reference spheroid.

```
% trajectory to Hadley Rille
eva1Lat = 26.15;
eva1Lon = 3.56;
[traj1Lat,traj1Lon] = track2(moduleLat,moduleLon,eva1Lat,eva1Lon,moonSpheroid);

% trajectory to Mons Hadley Delta hill complex
eva2Lat = 26.01;
eva2Lon = 3.59;
[traj2Lat,traj2Lon] = track2(moduleLat,moduleLon,eva2Lat,eva2Lon,moonSpheroid);
```

```
% trajectory to South Cluster hill complex
eva3Lat = 25.98;
eva3Lon = 3.697;
[traj3Lat,traj3Lon] = track2(moduleLat,moduleLon,eva3Lat,eva3Lon,moonSpheroid);
```

To facilitate plotting, collect the rover destination and trajectory variables for each EVA into combined variables. Use one variable each for these attributes: destination latitudes, destination longitudes, destination heights, trajectory latitudes, trajectory longitudes, and trajectory heights.

```
evaLat = [eva1Lat; eva2Lat; eva3Lat];
evaLon = [eva1Lon; eva2Lon; eva3Lon];
evaHeight = ones(length(evaLat),1) * roverHeight;

trajLat = [traj1Lat; traj2Lat; traj3Lat];
trajLon = [traj1Lon; traj2Lon; traj3Lon];
trajHeight = ones(length(trajLat),1) * roverHeight;
```

Specify the colors to use when plotting the trajectories, the lunar module, and the rover destinations. Use white for the trajectories, red for the lunar module, yellow for the rover destinations, and black for the marker outlines.

```
trajColor = "w";
moduleColor = "r";
roverColor = "y";
outlineColor = "k";
```

Visualize Terrain and Viewshed in 2-D

Visualize the terrain for the region and the viewshed from the lunar module by using `axesm`-based maps. `axesm`-based maps enable you to display geographic data using a map projection.

Visualize Terrain

Visualize the terrain for the region.

Create an `axesm`-based map by using an equidistant cylindrical projection and the Moon reference spheroid. The equidistant cylindrical projection is a reasonable choice for the region because it minimizes distortion near the equator. Set the latitude and longitude limits of the map using the latitude and longitude limits of the terrain data. Customize the appearance of the map by setting the positions of the meridian and parallel labels, specifying the number of significant digits to include in the labels, specifying the color and line style of the grid, and removing the frame. Reduce the white space around the map by using the `tightmap` function.

```
figure
axesm("eqdcylin",Geoid=moonSpheroid, ...                                % projection and spheroid
    MapLatLimit=R.LatitudeLimits,MapLonLimit=R.LongitudeLimits, ...     % latitude and longitude li
    MeridianLabel="on",MLineLocation=0.5,MLabelParallel="south", ...    % meridian label position
    ParallelLabel="on",PLineLocation=0.5, ...                           % parallel label position
    MlabelRound=-1,PlabelRound=-1, ...                                  % significant digits for la
    Grid="on",GColor=[0.8 0.8 0.8],GLineStyle="-.",GAltitude=Inf, ...   % grid appearance
    Frame="off")                                                        % frame
tightmap
```

Display the terrain data on a map using a surface. Add a labeled color bar.

```
geoshow(Z,R,DisplayType="texturemap")
cb = colorbar;
cb.Label.String = "Elevation in meters (with respect to reference spheroid)";
```

Display the trajectories on the map using white circle markers.

```
geoshow(trajLat,trajLon,DisplayType="point",ZData=trajHeight, ...
    Marker="o",MarkerSize=1,MarkerFaceColor=trajColor,MarkerEdgeColor=trajColor)
```

Display the lunar module on the map using a red circle marker. Prepare to include the module in a legend by specifying a display name and setting the plot object to a variable.

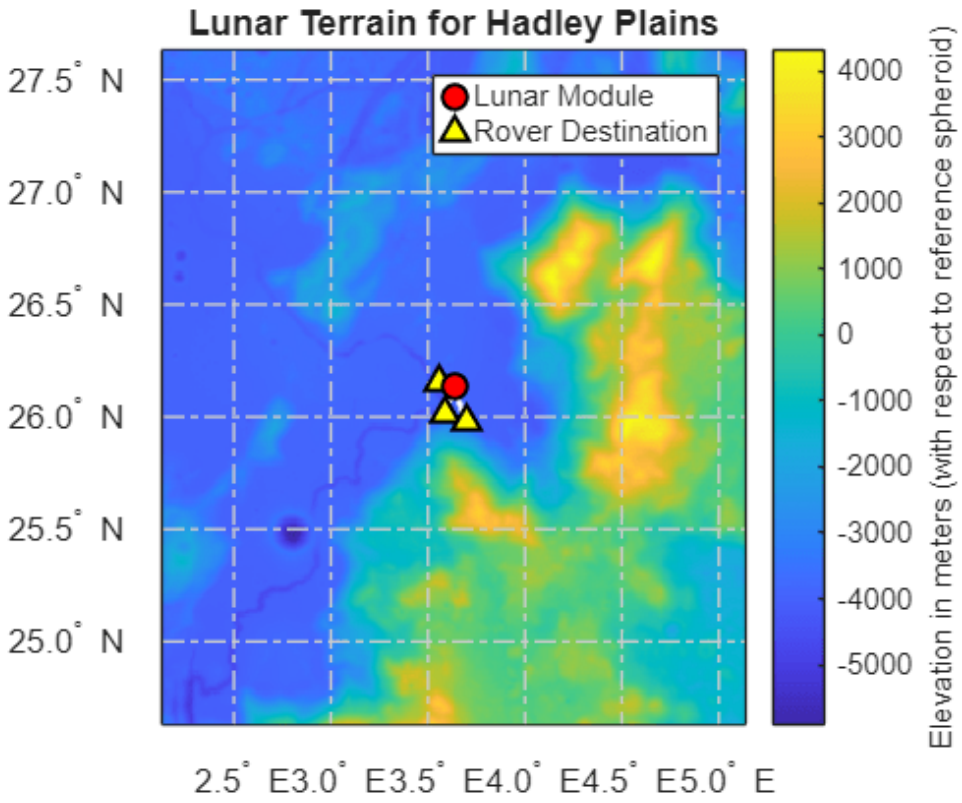
```
modulePt = geoshow(moduleLat,moduleLon,DisplayType="point",ZData=moduleHeight, ...
    Marker="o",MarkerSize=7.5,MarkerFaceColor=moduleColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Lunar Module");
```

Display the rover destinations on the map using yellow triangle markers. Prepare to include the destinations in the legend by specifying a display name and setting the plot object to a variable.

```
roverPt = geoshow(evaLat,evaLon,DisplayType="point",ZData=evaHeight, ...
    Marker="^",MarkerSize=7.5,MarkerFaceColor=roverColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Rover Destination");
```

Add a title. Include the lunar module and the rover destinations in a legend.

```
title("Lunar Terrain for Hadley Plains")
legend([modulePt roverPt])
```



Visualize Terrain for AOI

Visualize the terrain for only the area surrounding the lunar module by cropping the terrain data to a smaller region.

Specify an AOI by finding the coordinates of points that are 5 kilometers north, east, south, and west of the lunar module. Calculate the latitude and longitude limits of the AOI.

```
[boundinglat,boundinglon] = reckon(moduleLat,moduleLon,5000,[0; 90; 180; 270],moonSpheroid);
AOI = aoiquad(boundinglat,boundinglon);
[latlim,lonlim] = bounds(AOI);
```

Crop the terrain elevation data to the AOI. Resize the cropped data so that the array has 200 rows and 200 columns.

```
[ZAOI,RAOI] = geocrop(Z,R,latlim,lonlim);

gridSize = [200 200];
latscale = gridSize(1)/RAOI.RasterSize(1);
lonscale = gridSize(2)/RAOI.RasterSize(2);
[ZAOI,RAOI] = georesize(ZAOI,RAOI,latscale,lonscale);
```

Create a new `axesm`-based map. Then, display the terrain data on the map.

```
figure
axesm("eqdcylin",Geoid=moonSpheroid, ...                                    % projection and spher
    MapLatLimit=RAOI.LatitudeLimits,MapLonLimit=RAOI.LongitudeLimits, ... % latitude and longitu
    MeridianLabel="on",MLineLocation=0.1,MLabelParallel="south", ...       % meridian label posit
    ParallelLabel="on",PLineLocation=0.1, ...                              % parallel label posit
    MlabelRound=-1,PlabelRound=-1, ...                                     % significant digits f
    Grid="on",GColor=[0.8 0.8 0.8],GLineStyle="-.",GAltitude=Inf, ...      % grid appearance
    Frame="off")                                                           % frame
tightmap

geoshow(ZAOI,RAOI,DisplayType="texturemap")
```

Specify limits for the colormap using the minimum and maximum elevation values. Add a labeled color bar.

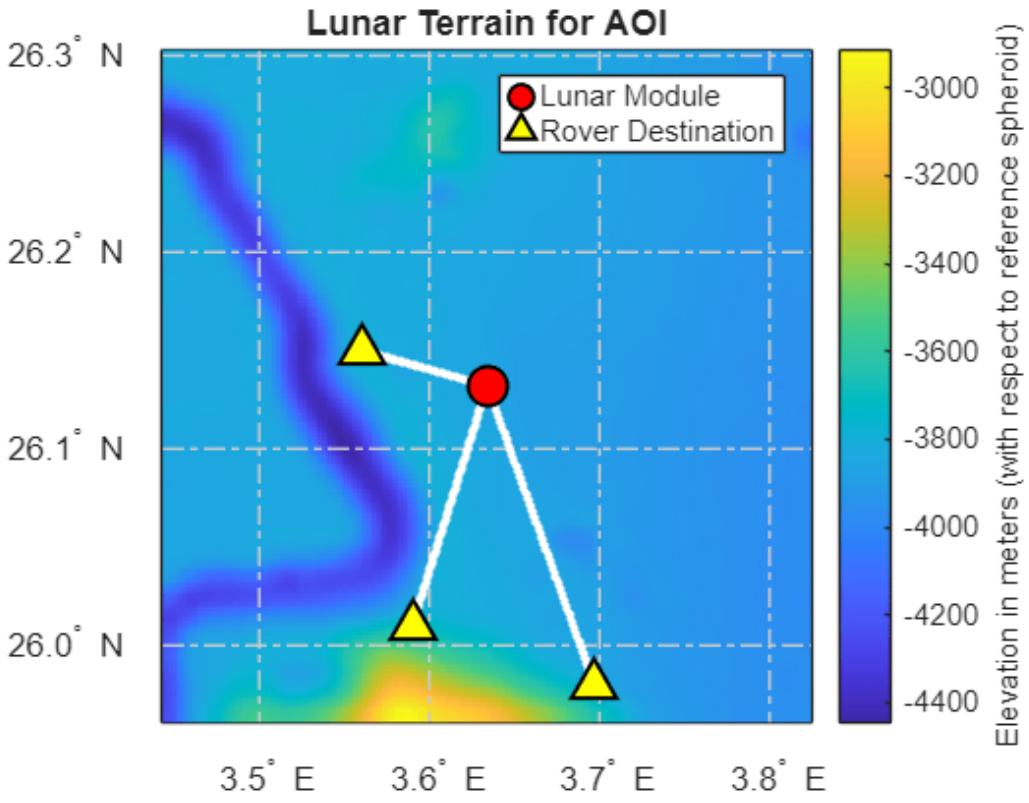
```
[cmin,cmax] = bounds(ZAOI,"all");
clim([cmin cmax])
cb = colorbar;
cb.Label.String = "Elevation in meters (with respect to reference spheroid)";
```

Display the trajectories, the lunar module, and the rover destinations on the map.

```
geoshow(trajLat,trajLon,DisplayType="point",ZData=trajHeight, ...
    Marker="o",MarkerSize=2,MarkerFaceColor=trajColor,MarkerEdgeColor=trajColor)

modulePt = geoshow(moduleLat,moduleLon,DisplayType="point",ZData=moduleHeight, ...
    Marker="o",MarkerSize=12,MarkerFaceColor=moduleColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Lunar Module");

roverPt = geoshow(evaLat,evaLon,DisplayType="point",ZData=evaHeight, ...
    Marker="^",MarkerSize=12,MarkerFaceColor=roverColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Rover Destination");
```

Add a title. Include the lunar module and the rover destinations in a legend.

```
legend([modulePt roverPt])
title("Lunar Terrain for AOI")
```



Visualize Viewshed for AOI

Calculate the viewshed for the lunar module. Indicate that the heights of the lunar module and the grid coordinates are referenced to the terrain by specifying the height references as `"AGL"`. Specify the radius of the reference sphere using the Moon spheroid.

```
visBase = viewshed(ZAOI,RAOI,moduleLat,moduleLon, ...
    moduleHeight,roverHeight,"AGL","AGL",moonSpheroid.Radius);
```

The `visBase` array returned by the `viewshed` function indicates visible areas using values of `1` and obscured areas using values of `0`. To avoid plotting the obscured areas, replace the `0` values with `NaN` values. Define a colormap for the viewshed that uses turquoise for visible areas.

```
visBase(visBase == 0) = NaN;
covcolor = [0.1034  0.8960  0.7150];
```

Create a new `axesm`-based map. Then, display the terrain data on the map.

```
figure
axesm("eqdcylin",Geoid=moonSpheroid, ...                                % projection and spher
    MapLatLimit=RAOI.LatitudeLimits,MapLonLimit=RAOI.LongitudeLimits, ... % latitude and longitu
    MeridianLabel="on",MLineLocation=0.1,MLabelParallel="south", ...    % meridian label posit
    ParallelLabel="on",PLineLocation=0.1, ...                           % parallel label posit
    MlabelRound=-1,PlabelRound=-1, ...                                  % significant digits f
    Grid="on",GColor=[0.8 0.8 0.8],GLineStyle="-.",GAltitude=Inf, ...   % grid appearance
    Frame="off")                                                        % frame
tightmap
```

```
geoshow(ZAOI,RAOI,DisplayType="texturemap")
```

Apply a gray colormap to the plotted terrain data. Specify the limits for the colormap using the minimum and maximum elevation values. Add a labeled color bar.

```
colormap("gray")
[cmin,cmax] = bounds(ZAOI,"all");
clim([cmin cmax])
cb = colorbar;
cb.Label.String = "Elevation in meters (with respect to reference spheroid)";
```

Display the viewshed for the lunar module over the terrain data. Prepare to include the viewshed in a legend by specifying a display name and setting the plot object to a variable.

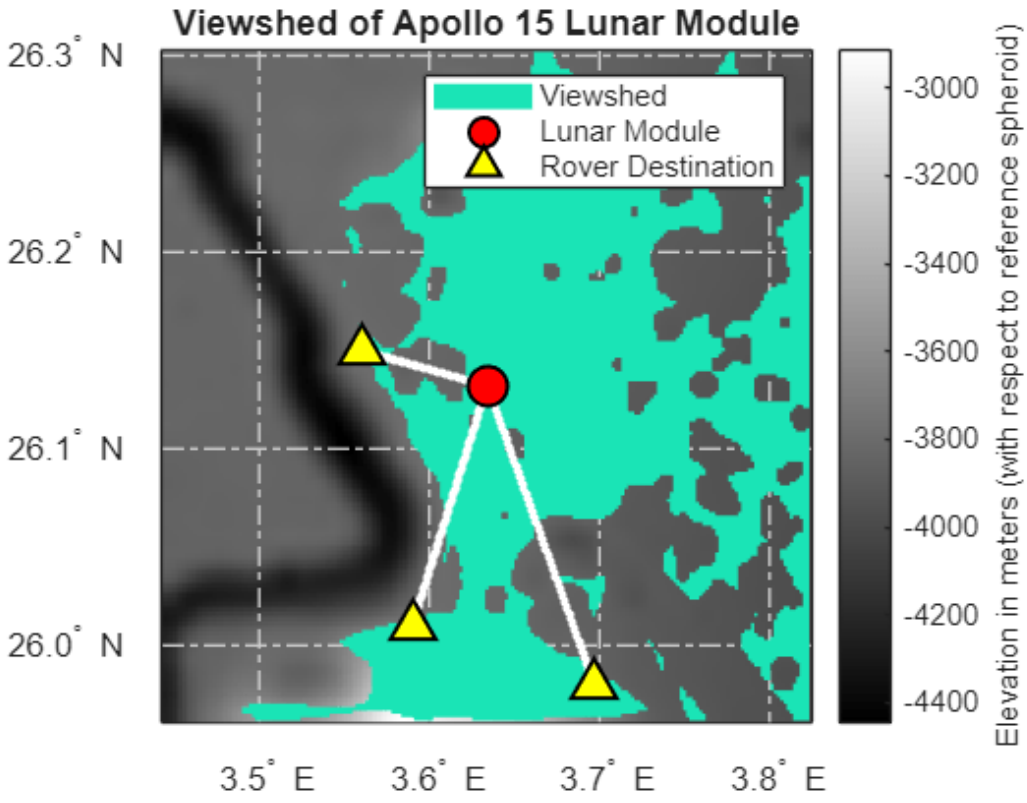
```
viewshedSurf = geoshow(visBase,RAOI,EdgeColor="none", ...
    DisplayType="surface",FaceColor=covcolor,DisplayName="Viewshed");
```

Display the trajectories, the lunar module, and the rover destinations on the map.

```
geoshow(trajLat,trajLon,DisplayType="point",ZData=trajHeight, ...
    Marker="o",MarkerSize=2,MarkerFaceColor=trajColor,MarkerEdgeColor=trajColor)

modulePt = geoshow(moduleLat,moduleLon,DisplayType="point",ZData=moduleHeight, ...
    Marker="o",MarkerSize=12,MarkerFaceColor=moduleColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Lunar Module");

roverPt = geoshow(evaLat,evaLon,DisplayType="point",ZData=evaHeight, ...
    Marker="^",MarkerSize=12,MarkerFaceColor=roverColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Rover Destination");
```

Add a title. Include the viewshed, the lunar module, and the rover destinations in a legend.

```
legend([viewshedSurf modulePt roverPt])
title("Viewshed of Apollo 15 Lunar Module")
```



Visualize Coverage in 2-D

A viewshed is comparable to a coverage map that includes only line-of-sight (LOS) propagation. You can create more detailed coverage maps and calculate received power by using antenna sites and ray tracing propagation models.

Create Antenna Sites

Prepare to create a coverage map by creating a transmitter site for the base station and a grid of receiver sites for the AOI.

To use a ray tracing propagation model with imported elevation data, you must specify the locations of the antenna sites using Cartesian coordinates. You can transform geographic coordinates to Cartesian coordinates in a local ENU system by using the `helperLatLonToENU` on page 11-57 helper function.

Create Transmitter Site

Specify the local origin of the ENU coordinate system using the geographic coordinates of the lunar module.

```
originlat = moduleLat;
originlon = moduleLon;
originht = 0;
```

Specify the geographic coordinates of the transmitter site. Then, transform the geographic coordinates to ENU coordinates by using the `helperLatLonToENU` helper function.

```
txlat = moduleLat;
txlon = moduleLon;
[txx,txy,txz] = helperLatLonToENU(Z,R,txlat,txlon,originlat,originlon,0,moonSpheroid);
```

Place the base station on top of the lunar module by adjusting the ENU *z*-coordinate.

```
txz = txz + moduleHeight;
```

Create the transmitter site. By default, transmitter sites use isotropic antennas.

```
tx = txsite("cartesian", ...
    AntennaPosition=[txx; txy; txz], ...
    TransmitterFrequency=moduleFreq, ...
    TransmitterPower=modulePower);
```

Create Receiver Sites

Extract a grid of geographic coordinates from the reference object for the AOI. Then, transform the geographic coordinates to local ENU coordinates by using the `helperLatLonToENU` helper function.

```
[latAOI,lonAOI] = geographicGrid(RAOI);
[rxx,rxy,rxz] = helperLatLonToENU(Z,R,latAOI(:),lonAOI(:), ...
    originlat,originlon,originht,moonSpheroid);
```

Place the receiver sites above the moon rover by adjusting the ENU *z*-coordinates.

```
rxz = rxz + roverHeight;
```

Create the receiver sites.

```
rxs = rxsite("cartesian",AntennaPosition=[rxx rxy rxz]');
```

Create Triangulation of Lunar Surface

Convert the terrain elevation data to a triangulation object. A triangulation object enables you to create coverage maps using ray tracing propagation model objects.

To use the ray tracing propagation model with a triangulation object, you must use a triangulation object referenced to Cartesian coordinates. Extract a grid of geographic coordinates from the reference object for the terrain data.

```
[gridlat,gridlon] = geographicGrid(R);
```

Convert the coordinates to Cartesian coordinates in an east-north-up (ENU) local coordinate system. Specify the origin of the ENU system using the coordinates of the lunar module.

```
[x,y,z] = geodetic2enu(gridlat,gridlon,Z,originlat,originlon,originht,moonSpheroid);
```

Convert the Cartesian coordinates to a patch with triangular faces. The `surf2patch` function represents the patch using two variables: `V` contains the patch vertices, and `F` determines which vertices to connect to form each triangular face. Prepare to visualize the triangulation object by reordering the vertices. Then, create the triangulation object from the patch.

```
[F,V] = surf2patch(x,y,z,"triangles");
F = F(:,[1 3 2]);
tri = triangulation(F,V);
```

Create Ray Tracing Propagation Model

Create a ray tracing propagation model, which MATLAB® represents using a `RayTracing` object. Specify the surface material using the real relative permittivity and the conductivity of the Moon surface [5] on page 11-57. By default, the model uses the shooting-and-bouncing rays (SBR) method.

```
epsR = 3.7;                                    % real relative permittivity of Moon surface
lossTan = 0.1;                                 % dielectric loss tangent
epsilon = 8.8541878128e-12;                    % permittivity of free space (vacuum)
cond = lossTan*(2*pi*moduleFreq)*epsR*epsilon; % conductivity of Moon surface

pm = propagationModel("raytracing",CoordinateSystem="cartesian", ...
    SurfaceMaterial="custom",SurfaceMaterialPermittivity=epsR, ...
    SurfaceMaterialConductivity=cond);
```

Visualize Line-of-Sight Coverage

Configure the propagation model to find LOS paths. Then, calculate the received power, in dBm, at each receiver site. Specify the terrain using the triangulation object.

```
pm.MaxNumReflections = 0;
sslos = sigstrength(rxs,tx,pm,Map=tri);
```

Prepare to display the coverage map.

- Specify power levels of interest between –120 dBm and –50 dBm.
- Define a colormap that uses 5 colors per 10 dBm.
- Specify colormap limits.
- Reshape the power values into a grid, and replace values below the minimum power threshold with `NaN` values.

```
levels = -120:10:-50;
cmap = turbo(5*length(levels));
climLevels = [min(levels)+20 max(levels)+30];

ss = sslos;
ss = reshape(ss,gridSize);
ss (ss < min(levels)) = NaN;
```

To display the terrain data and the received power values on the same map, use a tiled chart layout with one tile.

```
figure
t = tiledlayout(1,1);
```

Create an `axesm`-based map in the tile. Display the terrain data, and apply a gray colormap.

```
ax1 = axes(t);
axesm("eqdcylin",Geoid=moonSpheroid, ...                                    % projection and spher
    MapLatLimit=RAOI.LatitudeLimits,MapLonLimit=RAOI.LongitudeLimits, ... % latitude and longitu
    MeridianLabel="on",MLineLocation=0.1,MLabelParallel="south", ...       % meridian label posit
    ParallelLabel="on",PLineLocation=0.1, ...                              % parallel label posit
    MlabelRound=-1,PlabelRound=-1, ...                                     % significant digits fo
    Grid="on",GColor=[0.8 0.8 0.8],GLineStyle="-.",GAltitude=Inf, ...      % grid appearance
    Frame="off")                                                           % frame
tightmap
```

```
geoshow(ax1,ZAOI,RAOI,DisplayType="texturemap")
colormap(ax1,"gray")
```

Create a second `axesm`-based map in the same tile. Use the same projection, reference spheroid, and limits as the first map.

```
ax2 = axes(t);
abm2 = axesm("eqdcylin",Frame="off",Geoid=moonSpheroid, ...
    MapLatLimit=RAOI.LatitudeLimits,MapLonLimit=RAOI.LongitudeLimits);
abm2.Visible = "off";
tightmap
```

Display the received power values as a surface. Apply the power level colormap and set the color limits. Add a labeled color bar.

```
geoshow(ax2,ss,RAOI,DisplayType="surface")

colormap(ax2,cmap)
clim(ax2,climLevels)

cb = colorbar;
cb.Label.String = "Power (dBm)";
```

Display the trajectories, the lunar module, and the rover destinations in the second map.

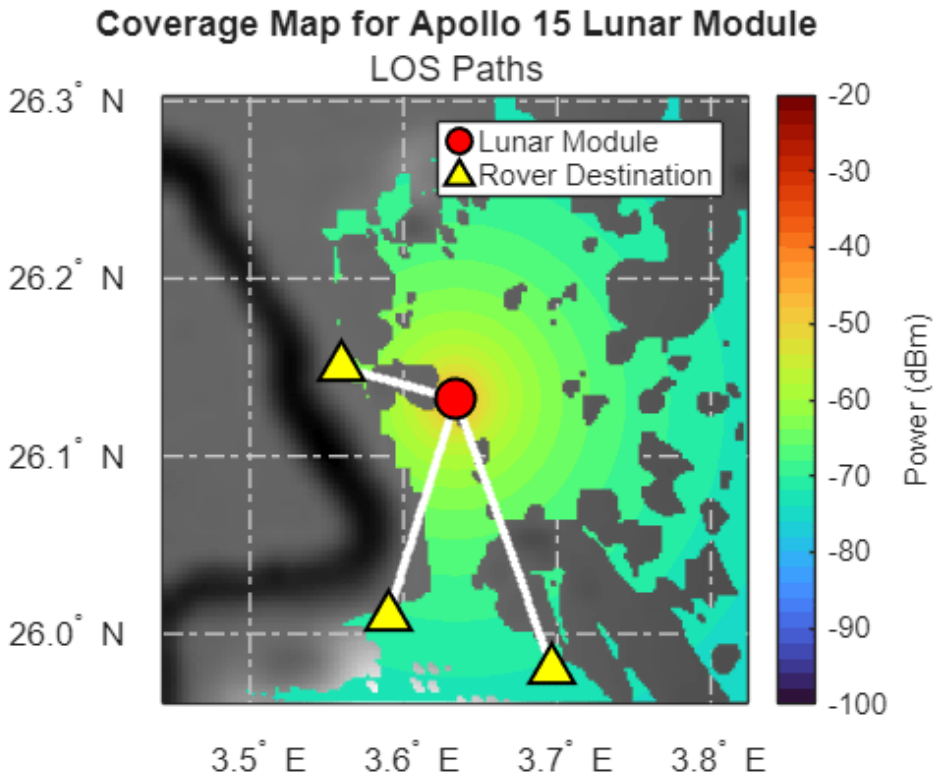
```
geoshow(trajLat,trajLon,DisplayType="point",ZData=trajHeight, ...
    Marker="o",MarkerSize=2,MarkerFaceColor=trajColor,MarkerEdgeColor=trajColor)

modulePt = geoshow(moduleLat,moduleLon,DisplayType="point",ZData=moduleHeight, ...
    Marker="o",MarkerSize=12,MarkerFaceColor=moduleColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Lunar Module");

roverPt = geoshow(evaLat,evaLon,DisplayType="point",ZData=evaHeight, ...
    Marker="^",MarkerSize=12,MarkerFaceColor=roverColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Rover Destination");
```

Link the axes objects that are stored in the tiles so that the axes objects share the same axis limits.

```
linkaxes([ax1 ax2],"xy")
```

Add a title and subtitle. Include the lunar module and the rover destinations in a legend.

```
title("Coverage Map for Apollo 15 Lunar Module",Visible="on")
subtitle("LOS Paths",Visible="on")
legend([modulePt roverPt])
```



Visualize Coverage with One Reflection

Configure the propagation model to find paths with up to one surface reflection. Then, recalculate the received power, in dBm, at each receiver site.

```
pm.MaxNumReflections = 1;
ssref = sigstrength(rxs,tx,pm,Map=tri);
```

Prepare to display the coverage map by reshaping the power values into a grid and by replacing values below the minimum threshold with `NaN` values.

```
ss = ssref;
ss = reshape(ss,gridSize);
ss (ss < min(levels)) = NaN;
```

Create a new tiled chart layout with one tile.

```
figure
t = tiledlayout(1,1);
```

Create an `axesm`-based map in the tile. Display the terrain data and apply a gray colormap.

```
ax1 = axes(t);
axesm("eqdcylin",Geoid=moonSpheroid, ...                                    % projection and spher
    MapLatLimit=RAOI.LatitudeLimits,MapLonLimit=RAOI.LongitudeLimits, ... % latitude and longitu
    MeridianLabel="on",MLineLocation=0.1,MLabelParallel="south", ...       % meridian label posit
    ParallelLabel="on",PLineLocation=0.1, ...                              % parallel label posit
    MlabelRound=-1,PlabelRound=-1, ...                                     % significant digits f
```

```
    Grid="on",GColor=[0.8 0.8 0.8],GLineStyle="-.",GAltitude=Inf, ...      % grid appearance
    Frame="off")                                                           % frame
tightmap

geoshow(ax1,ZAOI,RAOI,DisplayType="texturemap")
colormap(ax1,"gray")
```

Create a second `axesm`-based map in the same tile. Use the same projection, reference spheroid, and limits as the first map.

```
ax2 = axes(t);
abm2 = axesm("eqdcylin",Frame="off",Geoid=moonSpheroid, ...
    MapLatLimit=RAOI.LatitudeLimits,MapLonLimit=RAOI.LongitudeLimits);
abm2.Visible = "off";
tightmap
```

Display the received power values as a surface. Apply the power level colormap and set the color limits. Add a labeled color bar.

```
geoshow(ax2,ss,RAOI,DisplayType="surface")

colormap(ax2,cmap)
clim(ax2,climLevels)

cb = colorbar;
cb.Label.String = "Power (dBm)";
```

Display the trajectories, the lunar module, and the rover destinations in the second tile.

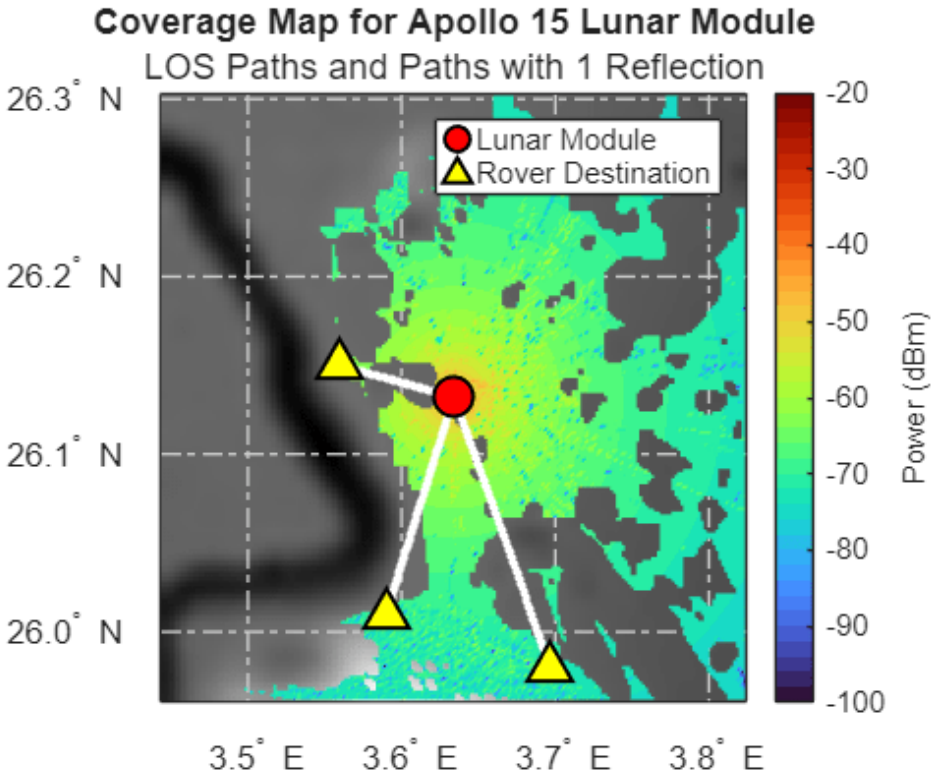
```
geoshow(trajLat,trajLon,DisplayType="point",ZData=trajHeight, ...
    Marker="o",MarkerSize=2,MarkerFaceColor=trajColor,MarkerEdgeColor=trajColor)

modulePt = geoshow(moduleLat,moduleLon,DisplayType="point",ZData=moduleHeight, ...
    Marker="o",MarkerSize=12,MarkerFaceColor=moduleColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Lunar Module");

roverPt = geoshow(evaLat,evaLon,DisplayType="point",ZData=evaHeight, ...
    Marker="^",MarkerSize=12,MarkerFaceColor=roverColor, ...
    LineWidth=1.2,MarkerEdgeColor=outlineColor,DisplayName="Rover Destination");
```

Link the axes objects that are stored in the tiles so that the axes objects share the same axis limits.

```
linkaxes([ax1 ax2],"xy")
```

Add a title and subtitle. Include the lunar module and the rover destinations in a legend.

```
title("Coverage Map for Apollo 15 Lunar Module",Visible="on")
subtitle("LOS Paths and Paths with 1 Reflection",Visible="on")
legend([modulePt roverPt])
```



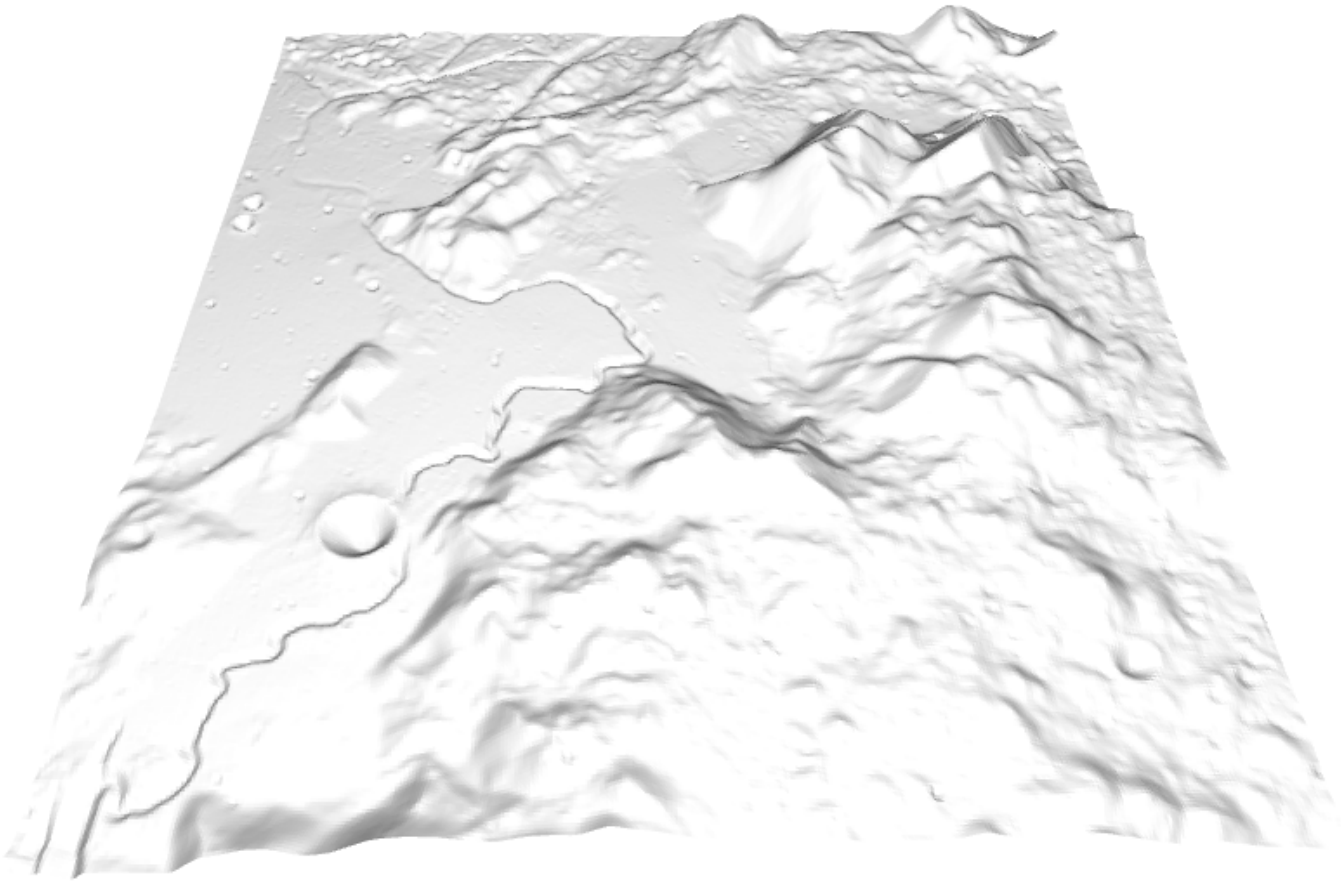
While the addition of paths with one reflection does not change the general shape of the coverage area, the interference between the line-of-sight paths and the reflected paths cause speckles in the power profile.

Visualize Terrain and Propagation Paths in 3-D

Visualize the terrain and propagation paths in 3-D using Site Viewer.

Import the triangulation object that represents the terrain model into Site Viewer. View the terrain for the AOI by zooming out and rotating the model.

```
sv = siteviewer(SceneModel=tri,ShowEdges=false,ShowOrigin=false);
```

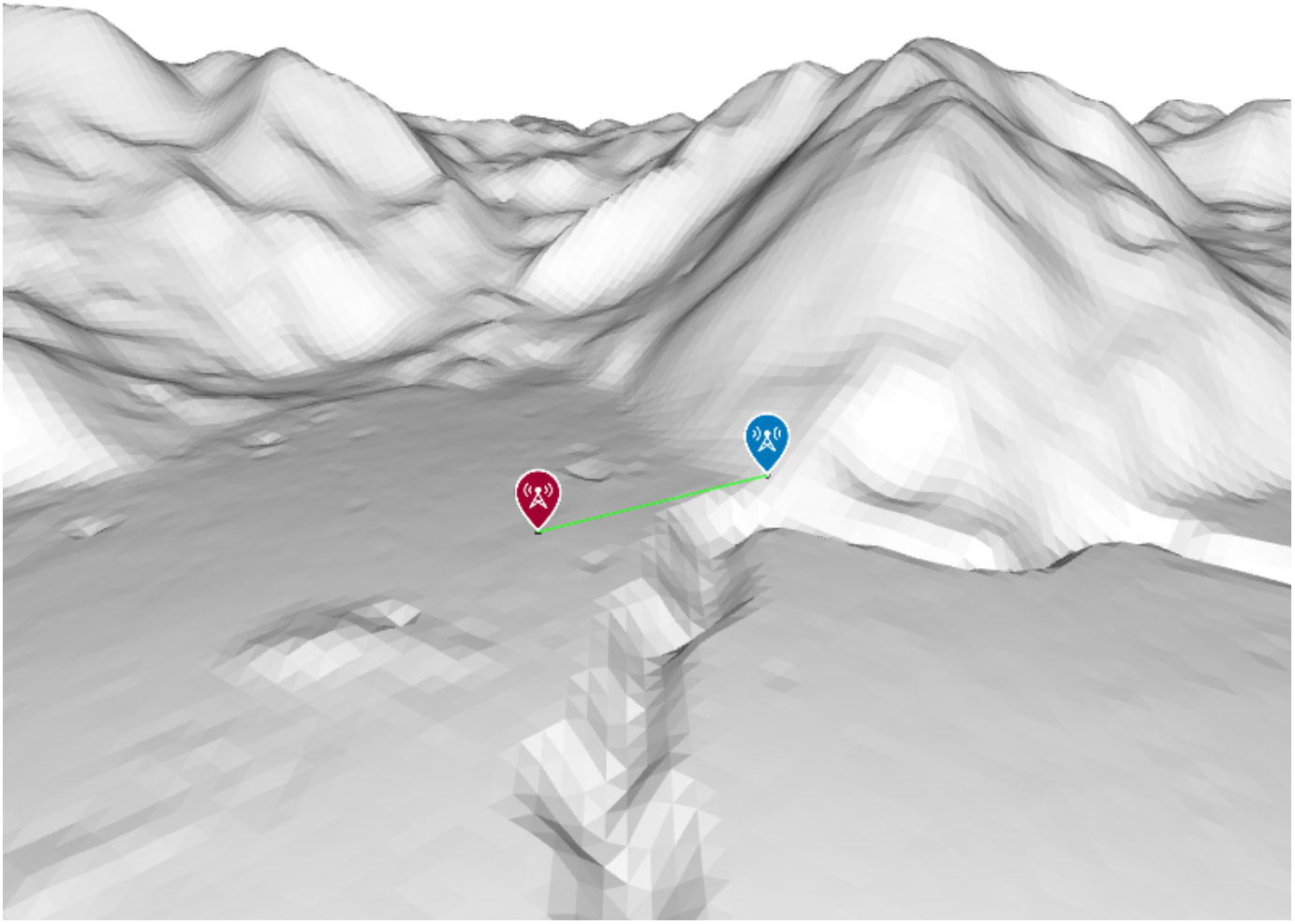


Create a receiver site that represents the destination for the EVA to the Mons Hadley Delta hill complex. Convert the geographic coordinates to local ENU coordinates by using the `helperLatLonToENU` function.

```
[rxx,rxy,rxz] = helperLatLonToENU(Z,R,eva2Lat,eva2Lon, ...
    originlat,originlon,originht,moonSpheroid);
rxz = rxz + roverHeight;
rx = rxsite("cartesian",AntennaPosition=[rxx rxy rxz]');
```

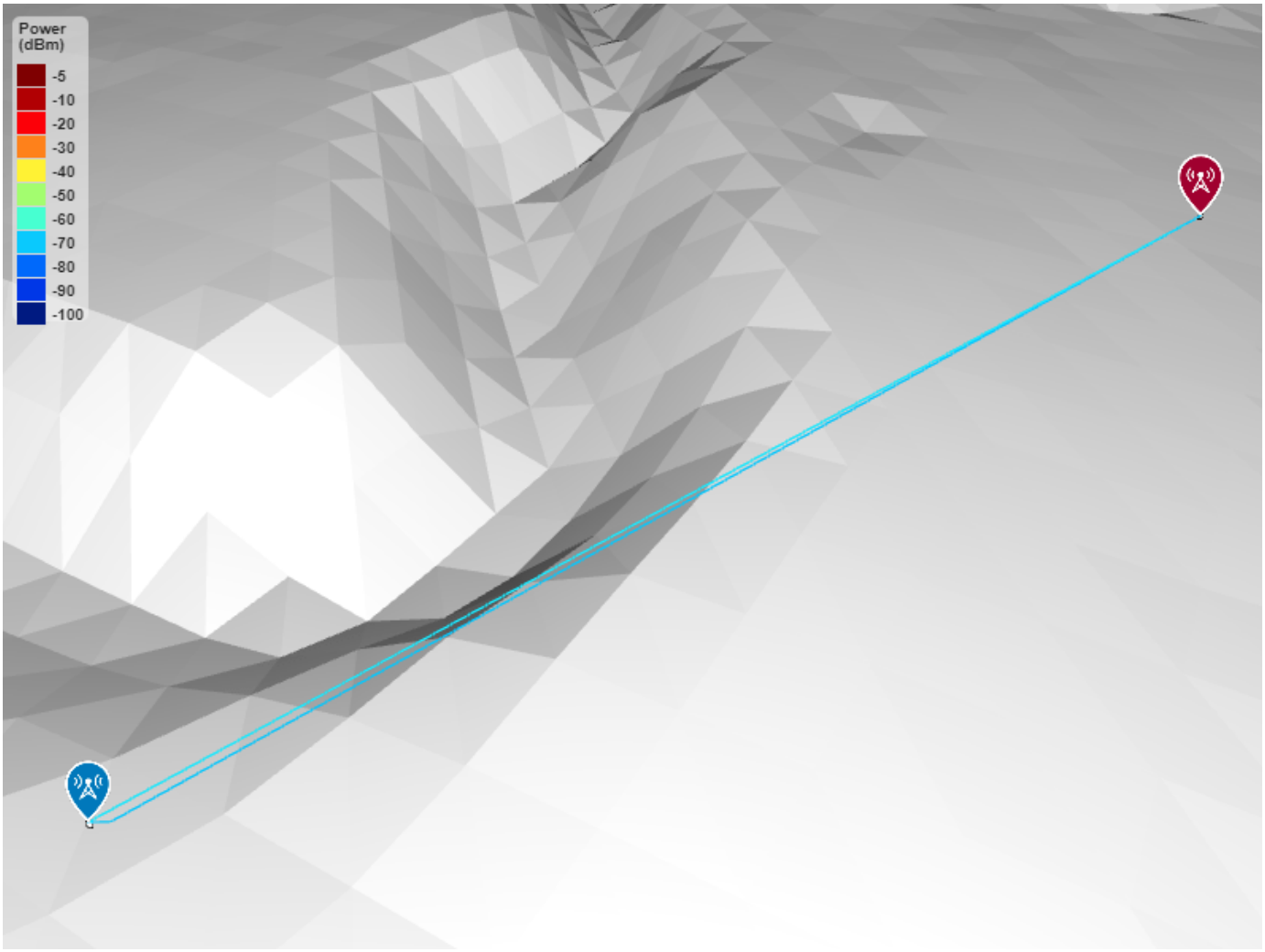
Display the transmitter site, the receiver site, and the LOS visibility status from the lunar module to the rover destination. The green line indicates that the rover is within LOS of the lunar module.

```
show(tx)
show(rx)
los(tx,rx)
```



Calculate and display the LOS and single-reflection propagation paths between transmitter and the receiver.

```
raytrace(tx,rx,pm,ColorLimits=[-100 -5])
```



References

[1] Barker, M.K., E. Mazarico, G.A. Neumann, M.T. Zuber, J. Haruyama, and D.E. Smith. "A New Lunar Digital Elevation Model from the Lunar Orbiter Laser Altimeter and SELENE Terrain Camera." *Icarus* 273 (July 2016): 346–55. https://doi.org/10.1016/j.icarus.2015.07.039.//doc/Bdoc/matlab/examples/custom_maps/main/internal/VisualizeCoverageMapsUsingRayTracingOverLunarTerrainExample.mlx

[2] NASA's Planetary Geology, Geophysics and Geochemistry Laboratory. "PGDA - High-resolution Lunar Topography (SLDEM2015)." NASA Goddard Space Flight Center. Accessed August 28, 2023. https://pgda.gsfc.nasa.gov/products/54.

[3] Davies, Merton E., and Tim R. Colvin. "Lunar coordinates in the regions of the Apollo landers." *Journal of Geophysical Research: Planets* 105, no. E8 (2000): 20277–80. https://doi.org/10.1029/1999JE001165.

[4] Peltzer, K.E. “Apollo Unified S-Band System.” Technical Memorandum (TM). Greenbelt, Maryland: NASA Goddard Space Flight Center, April 1, 1966. https://ntrs.nasa.gov/citations/19660018739.

[5] Heiken, Grant, David Vaniman, and Bevan M. French, eds. *Lunar Sourcebook: A User’s Guide to the Moon*. Cambridge [England]; New York: Cambridge University Press, 1991.

Helper Functions

The `helperLatLonToENU` helper function converts the geographic coordinates `lat` and `lon` along the terrain specified by `Zgeo` and `Rgeo` to local ENU coordinates. The input arguments `lat0`, `lon0`, and `h0` specify the origin of the ENU system. The input argument `spheroid` specifies the reference spheroid for the geographic coordinates.

```
function [x,y,z] = helperLatLonToENU(Zgeo,Rgeo,lat,lon,lat0,lon0,h0,spheroid)
% Convert geographic coordinates to local ENU coordinates
    h = geointerp(Zgeo,Rgeo,lat,lon);
    [x,y,z] = geodetic2enu(lat,lon,h,lat0,lon0,h0,spheroid);
end
```

See Also

Functions

`readgeoraster` | `geodetic2enu` | `sigstrength` | `raytrace`

Objects

`siteviewer` | `RayTracing`

Related Examples

- “Visualize Viewsheds and Coverage Maps Using Terrain” on page 11-9

Radar Performance Analysis over Terrain

The performance of a radar system is highly dependent on the environment in which it operates. While the free space spreading loss might be such that the target signal-to-noise ratio (SNR) does not satisfy the minimum detectability threshold for the desired probability of detection and probability of false alarm, the detectability of a target might further suffer in some terrains as there might be no direct, unobstructed line-of-sight from the radar to the target. As you will see in this example, as the elevation of a target increases above the terrain, the radar has a better chance of detecting the target.

In this example, you will learn how to analyze the performance of a ground-based, long-range terminal airport surveillance radar tasked with detecting an aircraft in the presence of heavy, mountainous clutter. The example first defines the radar system operating characteristics and its global position. It then defines the target and its trajectory. Finally, the detectability of the target as it moves through its trajectory is presented along with detailed visualizations.

Define Radar

To start, specify a C-band long-range, terminal airport surveillance radar with the following parameters:

- Peak power: 1 kW
- Operating frequency: 6 GHz
- Transmit and receive antenna: 2 degrees in azimuth, 5 degrees in elevation
- Pulse width: 1 µs

```
rdrppower = 1e3;            % Peak power (W)
fc = 6e9;                   % Operating frequency (Hz)
hpbw = [2; 5];              % Half-power beamwidth [azimuth; elevation] (deg)
rdrpulsew = 1e-6;           % Pulse width (s)
lambda = freq2wavelen(fc);  % Wavelength (m)
```

Convert the transmitter half-power beamwidth (HPBW) values to gain using the `beamwidth2gain` function. Assume a cosine rectangular aperture, which is a good approximation for a real-world antenna.

```
rdrgain = beamwidth2gain(hpbw,"CosineRectangular"); % Transmitter and receiver gain (dB)
```

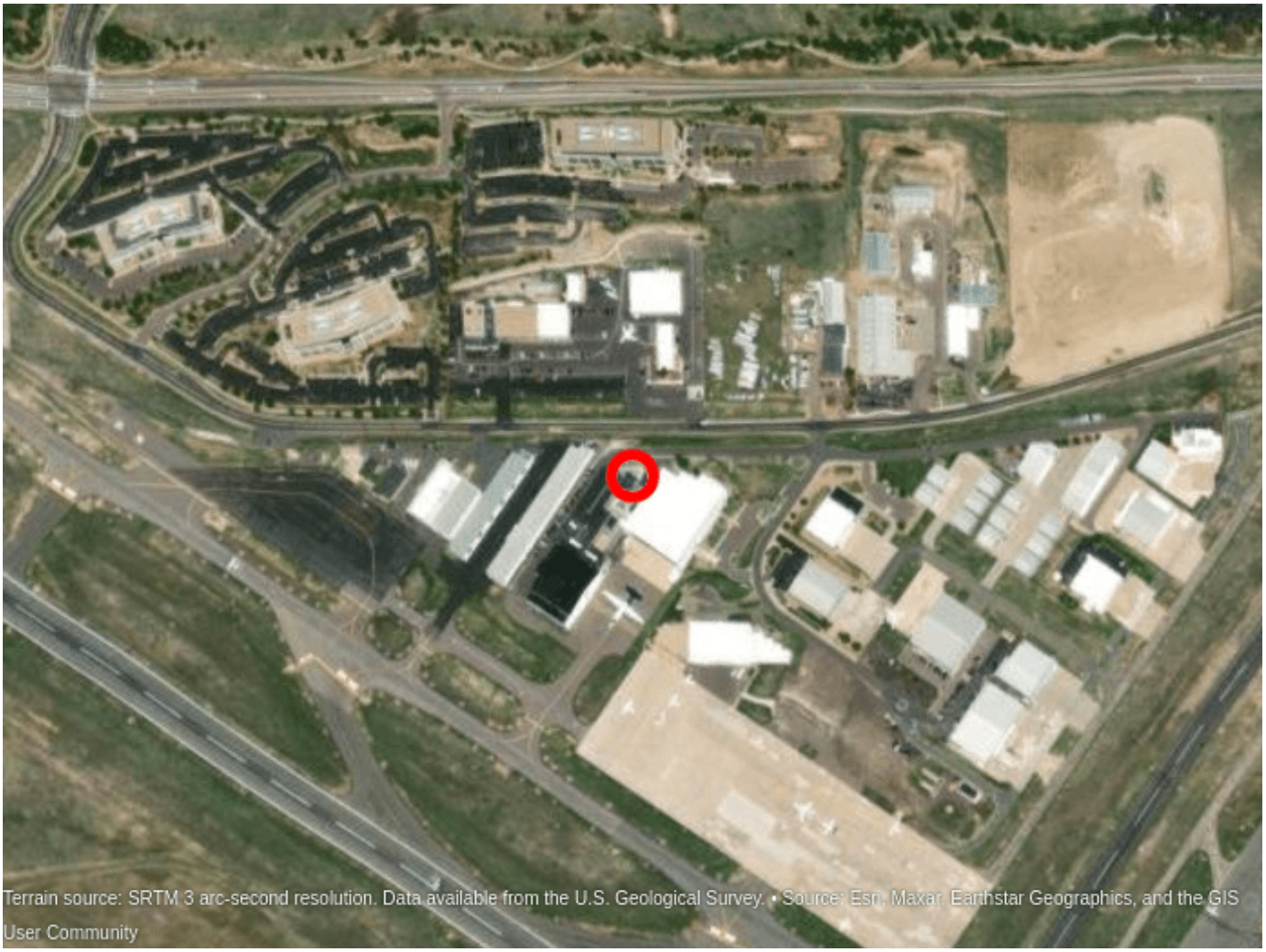
Define the radar ground location as the Rocky Mountain Metropolitan Airport in Broomfield, Colorado, USA. The radar is mounted on a tower 10 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height referenced to the mean sea level (MSL).

```
rdrlat = 39.913756;         % Radar latitude (deg)
rdrlon = -105.118062;       % Radar longitude (deg)
rdrtowerht = 10;            % Antenna height (m)
rdralt = 1717 + rdrtowerht; % Radar altitude (m)
```

To visualize the radar location, import the relevant terrain data from the United States Geological Survey (USGS). The input height to `geoplot3` is referenced to the geoid, or mean sea level.

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.";
[Zterrain,Rterrain] = readgeoraster(dtedfile,"OutputType","double");
```

```
% Visualize the location using the geographic globe plot.
addCustomTerrain("southboulder",dtedfile,Attribution=attribution);
fig = uifigure;
g = geoglobe(fig,Terrain="southboulder");
hold(g,"on")
h_rdrtraj = geoplot3(g,rdrlat,rdrlon,rdralt,"ro",LineWidth=6,MarkerSize=10);
```



Note that the limits for the file correspond to the region around Boulder, Colorado, USA, and the resolution corresponds to DTED level-1, which has a sample resolution of 3 arc seconds or about 90 meters.

Define Target

Consider a large commercial aircraft as the target. Assume the aircraft trajectory is a corkscrew maneuver, where the aircraft descends rapidly in a spiral.

```
tlat0 = 39.80384;           % Target initial latitude (deg)
tlon0 = -105.49916;         % Target initial longitude (deg)
tht0 = 3000;                % Target initial height (m)
azs = 1:2:540;              % Target azimuth (deg)
```

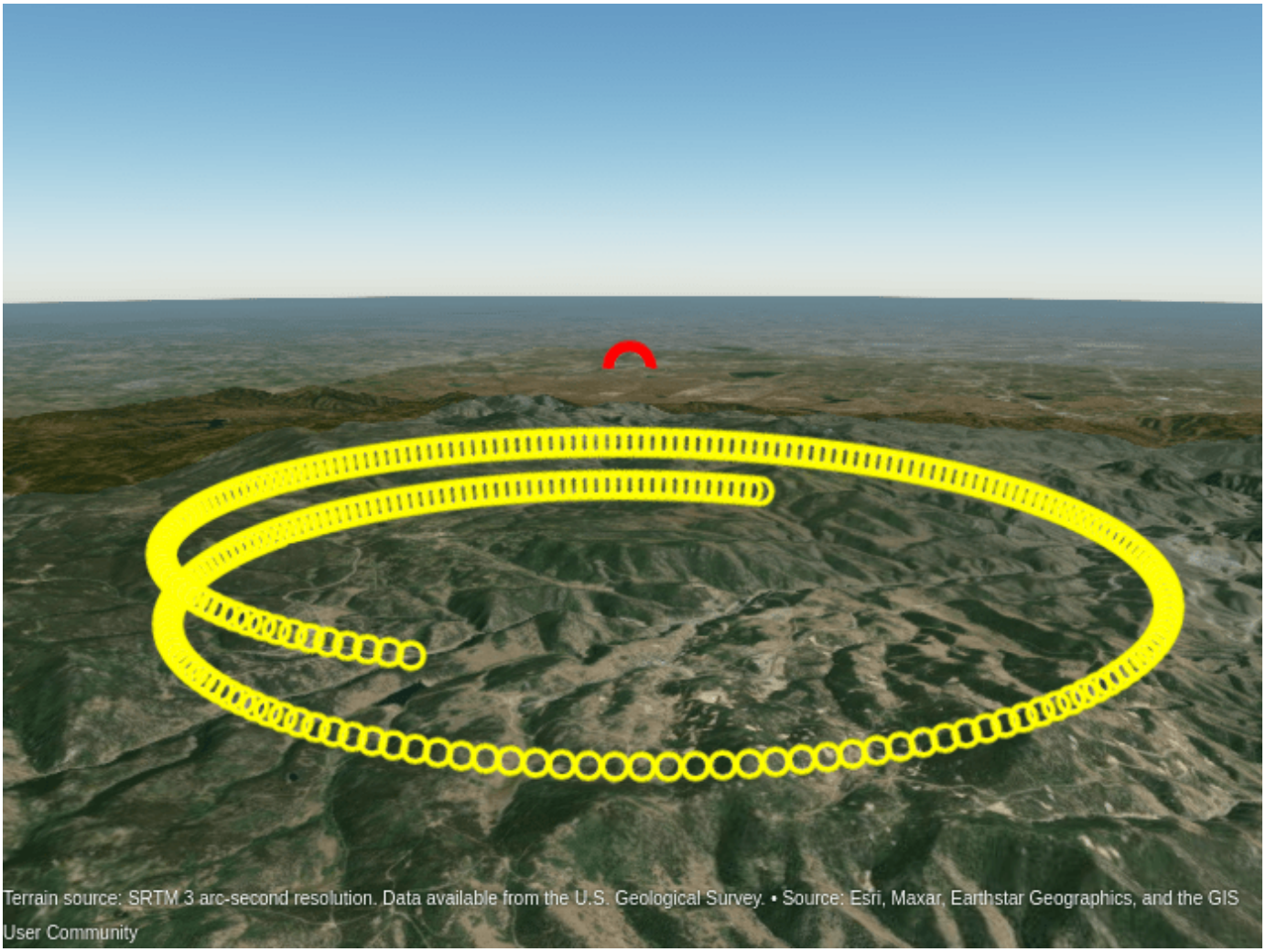
```
r = 5000;                   % Target slant range (m)

% Convert from polar coordinates to Cartesian East, North, Up (ENU).
[X,Y] = pol2cart(deg2rad(azs),r);

% Convert ENU to geodetic.
Z = linspace(0,1000,numel(azs));
wgs84 = wgs84Ellipsoid;
[tlat,tlon,tht] = enu2geodetic(X,Y,Z,tlat0,tlon0,tht0,wgs84);

% Define the target altitude relative to the geoid.
talt = tht - egm96geoid(tlat,tlon); % Target altitude (m)
```

For simplicity, assume the waypoints are obtained at a constant sampling rate of 0.1 Hz. The trajectory can be generated using `geoTrajectory` with positions specified as latitude, longitude, and height. Height for `geoTrajectory` is referenced to the WGS84 ellipsoid.

```
fs = 0.1;
t = (0:length(X)-1)/fs;
ttraj = geoTrajectory([tlat.' tlon.' tht.'],t,SampleRate=fs);
```

Plot the ground truth trajectory over the terrain.

```
h_ttraj = geoplot3(g,tlat,tlon,talt,"yo",LineWidth=3);
campos(g,39.77114,-105.62662,6670)
camheading(g,70)
campitch(g,-12)
```



The radar cross section (RCS) for an aircraft is typically from 1 to 10 square meters. For this example, consider the aircraft as an isotropic point target with an RCS of 10 square meters.

```
trcs = pow2db(10);          % Target RCS (dBsm)
```

Simulate the Scenario

Now that the radar and target have been defined, build the scenario, which is composed of a terminal airport radar and large commercial aircraft in the presence of mountainous clutter. Run the simulated scenario for the duration of the aircraft trajectory.

```
scene = radarScenario(IsEarthCentered=true,UpdateRate=fs,StopTime=t(end));
rdrplatform = platform(scene,Position=[rdrlat,rdrlon,rdralt],Sensor=radarDataGenerator);
tplatform = platform(scene,Trajectory=ttraj,Signatures= ...
    {rcsSignature(Azimuth=[-180 180],Elevation=[-90 90],Pattern=trcs)});
```

The line-of-sight path from the radar to the target is determined for each point in the target trajectory. For locations where the aircraft is not occluded by the terrain, an SNR value is calculated using the radar equation, including the propagation factor along the path.

The propagation factor is calculated using the `radarpropfactor` function. The default permittivity model in `radarpropfactor` is based on a sea permittivity model in Blake's *Machine Plotting of Radar Vertical-Plane Coverage Diagrams*. Such a model is not applicable in this example. Thus, the first step in simulating more realistic propagation is to select a more appropriate permittivity. Use the `earthSurfacePermittivity` function with the vegetation flag. Assume an ambient temperature of 21.1 degrees Celsius, which is about 70 degrees Fahrenheit. Assume a gravimetric water content of 0.3.

```
temp = 21.1;                % Ambient temperature (degrees Celsius)
gwc = 0.3;                  % Gravimetric water content
[~,~,epsc] = earthSurfacePermittivity("vegetation",fc,temp,gwc);
```

Calculate the propagation factor using the `radarpropfactor` function. Include the following in the calculation:

- Surface permittivity
- Standard deviation of height along the path
- Elevation beamwidth

```
tsnr = -inf(size(t));
F = zeros(size(t));
trange = zeros(size(t));
isVisible = false(size(t));
idx = 1;

while advance(scene)
    tpose = pose(tplatform,CoordinateSystem="Geodetic");
    tpos = tpose.Position;
    [isVisible(idx),~,~,h] = los2(Zterrain,Rterrain,rdrlat,rdrlon, ...
        tpos(1),tpos(2),rdralt,tpos(3),"MSL","MSL");
    hgtStdDev = std(h);
    if isVisible(idx)
        trange(idx) = llarangeangle([rdrlat rdrlon rdralt],tpos,1);
        F(idx) = radarpropfactor(trange(idx),fc,rdralt,tpos(3), ...
            SurfaceRelativePermittivity=epsc,...
            SurfaceHeightStandardDeviation=hgtStdDev, ...
            ElevationBeamwidth=hpbw(2));
    end
    idx = idx+1;
end
```

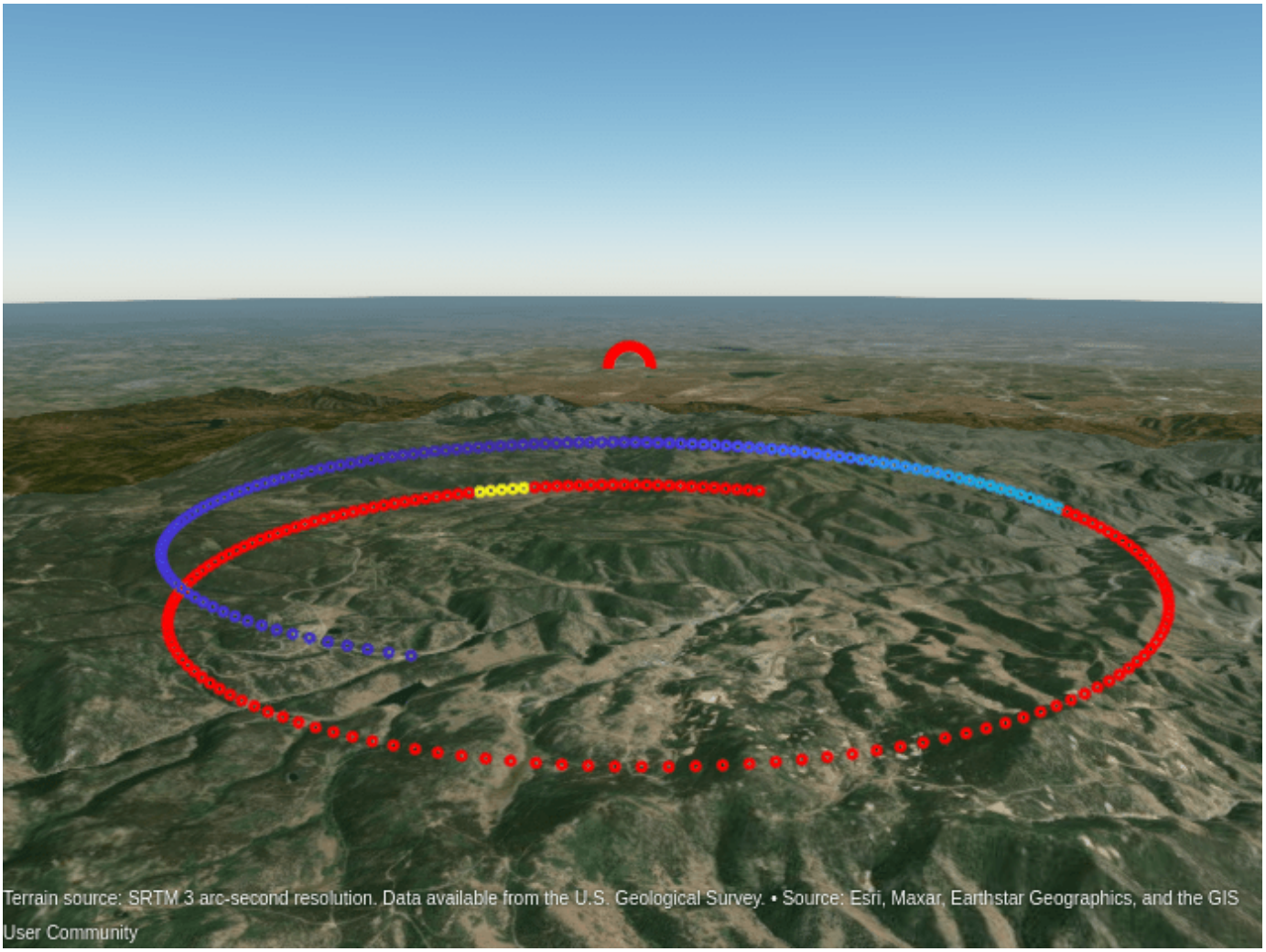
Calculate the SNR along the trajectory.

```
tsnr(isVisible) = radareqsnr(lambda,trange(isVisible).',rdrppower,rdrpulsew,...
    RCS=trcs,Gain=rdrgain,PropagationFactor=F(isVisible).');
```

Next, plot the SNR along the trajectory.

```
tsnr_finiteidx = ~isinf(tsnr);
tsnr_cidx = zeros(size(tsnr));
cmap = colormap(g);
numclvls = size(cmap,1);
tsnr_cidx(tsnr_finiteidx) = discretize(tsnr(tsnr_finiteidx),numclvls-1);
tsnr_cidx(~tsnr_finiteidx) = numclvls;

delete(h_ttraj);
hsnr = zeros(size(tsnr));
```

```
for m = 1:numel(tsnr)
    hsnr(m) = geoplot3(g,tlat(m),tlon(m),talt(m), ...
        Marker="o",LineWidth=2,MarkerSize=1);
    if tsnr_finiteidx(m)
        set(hsnr(m),"Color",cmap(tsnr_cidx(m),:));
    else
        set(hsnr(m),"Color","r");
    end
end
```



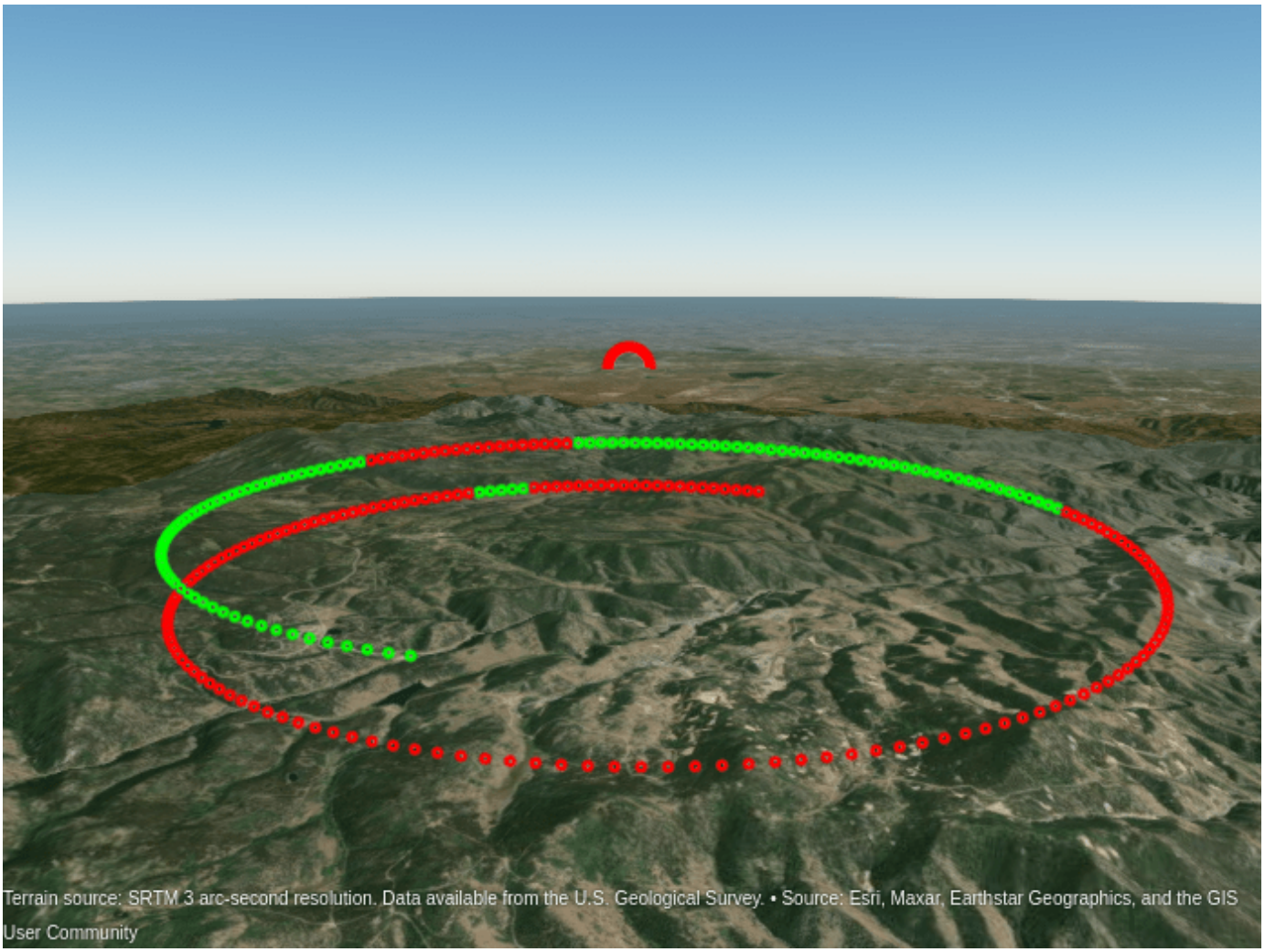
As the aircraft performs the corkscrew maneuver, the SNR of the received signal might vary, as shown in the figure. The radar has an unobstructed view of the aircraft if there is a line-of-sight path. The red portion of the trajectory indicates that there is no line-of-sight path between the aircraft and the radar.

For the surveillance radar, the desired performance index is a probability of detection (Pd) of 0.9 and probability of false alarm (Pfa) below 1e-6. To make the radar system design more feasible, you can use a pulse integration technique to reduce the required SNR. For this system, assume noncoherent integration of 32 pulses. A good approximation of the minimum SNR needed for a detection at the specified Pd and Pfa can be computed by the `detectability` function.

```
pd = 0.9;
pfa = 1e-6;
minsnr_32p = detectability(pd,pfa,32);
isdetectable_32p = tsnr >= minsnr_32p;
```

Observe in which part of the trajectory the target is detectable (shown in green) given the minimum SNR requirement. Note that the existence of a line-of-sight link does not guarantee that the target is detectable.

```
for m = 1:numel(tsnr)
    if isdetectable_32p(m)
        set(hsnr(m),Color="g");
    else
        set(hsnr(m),Color="r");
    end
end
```



To improve detectability with surveillance radars, often radar engineers discuss maximizing the power-aperture product of the system. This generally translates to increasing the physical size or peak power of a system. It can also be considered in terms of illumination time (i.e., energy on target). Some methods to improve detectability include:

- Increasing the peak power: This can be difficult to achieve due to limitations on the power supply and on the radar platform location. Furthermore, if there is a requirement for low probability of intercept (LPI), increasing peak power is often undesirable.
- Increasing the physical size of the antenna aperture: Increasing the physical size of the antenna results in an increase in the associated gain and decrease in the half-power beamwidth. Limitations of the platform or location might make increasing the physical size of the antenna aperture infeasible. Additionally, with a finer beamwidth, it becomes more important that the antenna beam is steered towards the target under test.
- Increasing the number of pulses to be integrated: This will bring the detectability to a lower value. However, if the aircraft is maneuvering at a high speed, then it might take too long to collect all of the transmitted pulses under the assumption of target stationarity. If the target stationarity assumption is invalid, additional signal processing steps will need to be taken to mitigate the range walk of the target.
- Increasing the average power: Rather than increasing the peak power, one can increase the average power by increasing the duty cycle. Increasing the duty cycle means increasing either the pulse width or the pulse repetition frequency (PRF), which might put undue burden on the radar hardware. The downside to increasing the pulse width is an increase in the minimum range and potentially overlapping, inseparable target returns. On the other hand, increasing the pulse repetition frequency decreases the maximum unambiguous range, which might be undesirable for a long-range surveillance system, especially if it is not performing a disambiguation technique.

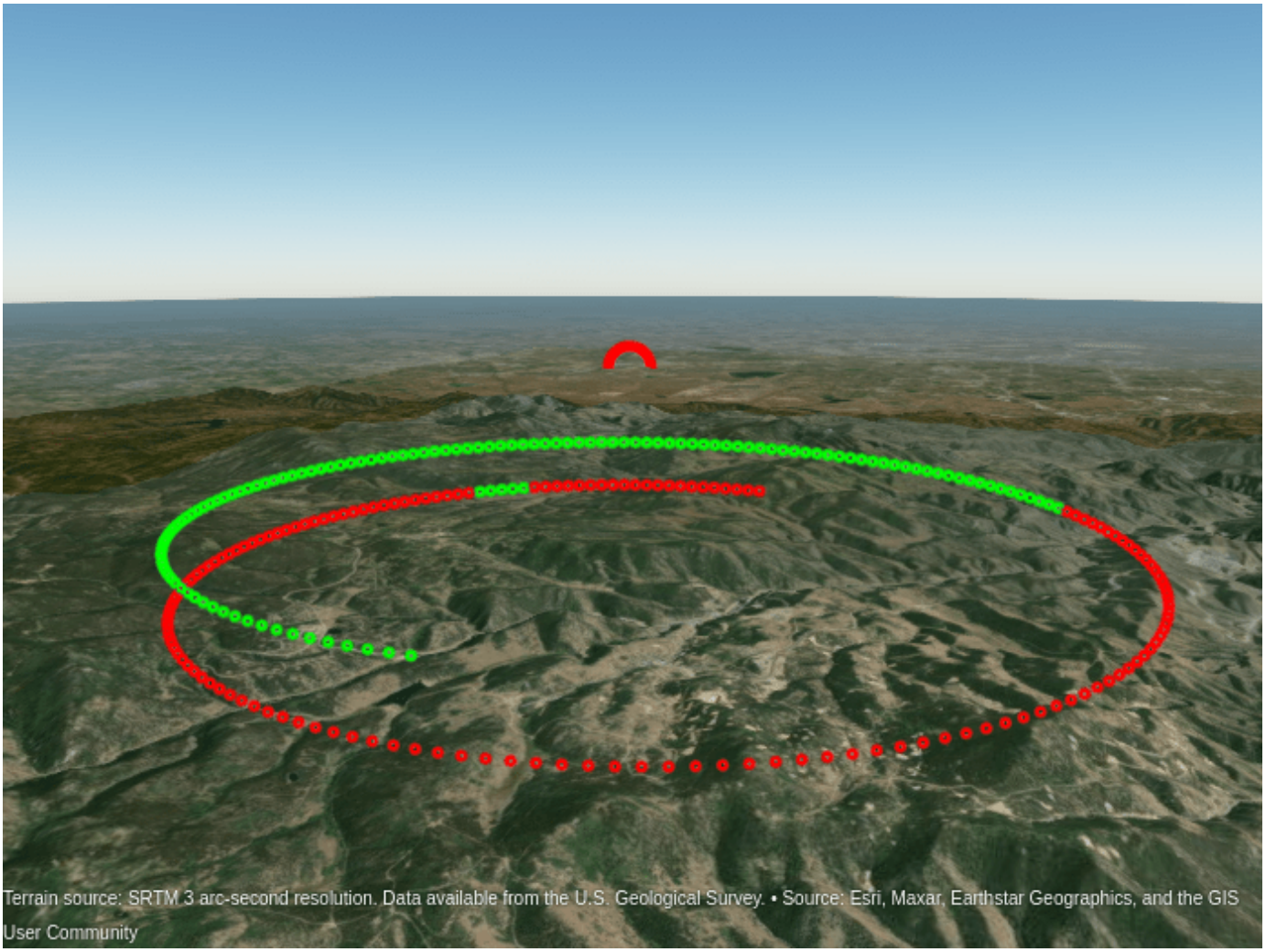
The above list, while in no way exhaustive, shows some of the tradeoffs in the design of an terminal airport surveillance system. For this example, increase the peak power. Since this is a ground-based system, increasing the power is not expected to be too burdensome. Additionally, other airport radars like the ASR-9 operate at a peak power of about 1 MW. Since this is an airport radar, there is no need for LPI requirements.

Consider the case where the peak power is increased to 10 kW.

```
rdrppower = 10e3;          % Peak power (W)

% Recalculate the SNR along the trajectory.
tsnr(isVisible) = radareqsnr(lambda,trange(isVisible).',rdrppower,rdrpulsew,...
    RCS=trcs,Gain=rdrgain,PropagationFactor=F(isVisible).');

% Determine the regions of the trajectory that are now detectable given the
% newly updated SNR.
isdetectable_32p = tsnr >= minsnr_32p;
```

Notice that by increasing the peak power, the regions at the end of the trajectory that were previously undetected now satisfy the minimum SNR threshold.

```
for m = 1:numel(tsnr)
    if isdetectable_32p(m)
        set(hsnr(m),Color="g");
    else
        set(hsnr(m),Color="r");
    end
end
```



Summary

In this example, SNR is calculated and visualized for a ground-based, long-range terminal airport surveillance radar tasked with detecting an aircraft in the presence of heavy, mountainous clutter. The example shows how to calculate the line-of-sight given a target trajectory. It also shows that the existence of a line-of-sight link does not necessarily guarantee that the target is detectable. This example considers some design tradeoffs to improve target detectability, discussing how radar parameters can be modified to match other system requirements. The example can be easily extended for other aircraft, different trajectory sets, and different terrain maps.

```
% Clean up by closing the geographic globe and removing the imported
% terrain data.
if isvalid(fig)
    close(fig)
```

```
end
removeCustomTerrain("southboulder")
```

See Also

Functions

`addCustomTerrain` | `geoglobe` | `geoplot3` | `radarScenario` | `radareqsnr`

Related Examples

- “Radar Vertical Coverage over Terrain” (Radar Toolbox)

Radar Vertical Coverage over Terrain

The propagation path of a radar system is modified by the environment in which it operates. When overlooking a surface, the free space radiation pattern is modified by the reflected wave, which results in an interference pattern with lobes and nulls. In a lobe maximum, a radar return can increase by as much as 12 dB, whereas in a minimum the return can theoretically go to 0 dB.

The vertical coverage pattern is a detection contour. It offers a look into the performance of a radar system at a constant signal level as specified by the free space range. The vertical coverage considers the interference between the direct and ground-reflected rays, as well as refraction. This example defines an L-band radar system in the presence of heavy clutter and shows you how to visualize its three-dimensional vertical coverage over terrain.

Define the Radar

To start, specify an L-band surveillance radar with the following parameters:

- Peak power: 3 kW
- Operating frequency: 1 GHz
- Transmit and receive antenna beamwidth: 20 degrees in azimuth and elevation
- Pulse width: 2 µs

```
% Radar parameters
rfs        = 50e3;           % Free-space range (m)
rdrppower  = 3e3;            % Peak power (W)
fc         = 1e9;            % Operating frequency (Hz)
hpbw       = 20;             % Half-power beamwidth (deg)
rdrpulsew  = 2e-6;           % Pulse width (s)
tiltAng    = 1;              % Radar tilt (elevation) angle (deg)
azRotation = 80;             % Radar azimuth rotation angle (deg)

% Vertical coverage parameters
minEl      = 0;              % Minimum vertical coverage angle (deg)
maxEl      = 90;             % Maximum vertical coverage angle (deg)
elStepSize = 1;              % Elevation step size (deg)
azAng      = -60:2:60;       % Azimuth angles for elevation cuts (deg)
```

Convert the transmit antenna's half-power beamwidth (HPBW) values to gain using the `beamwidth2gain` function. Assume a cosine rectangular aperture, which is a good approximation for a real-world antenna.

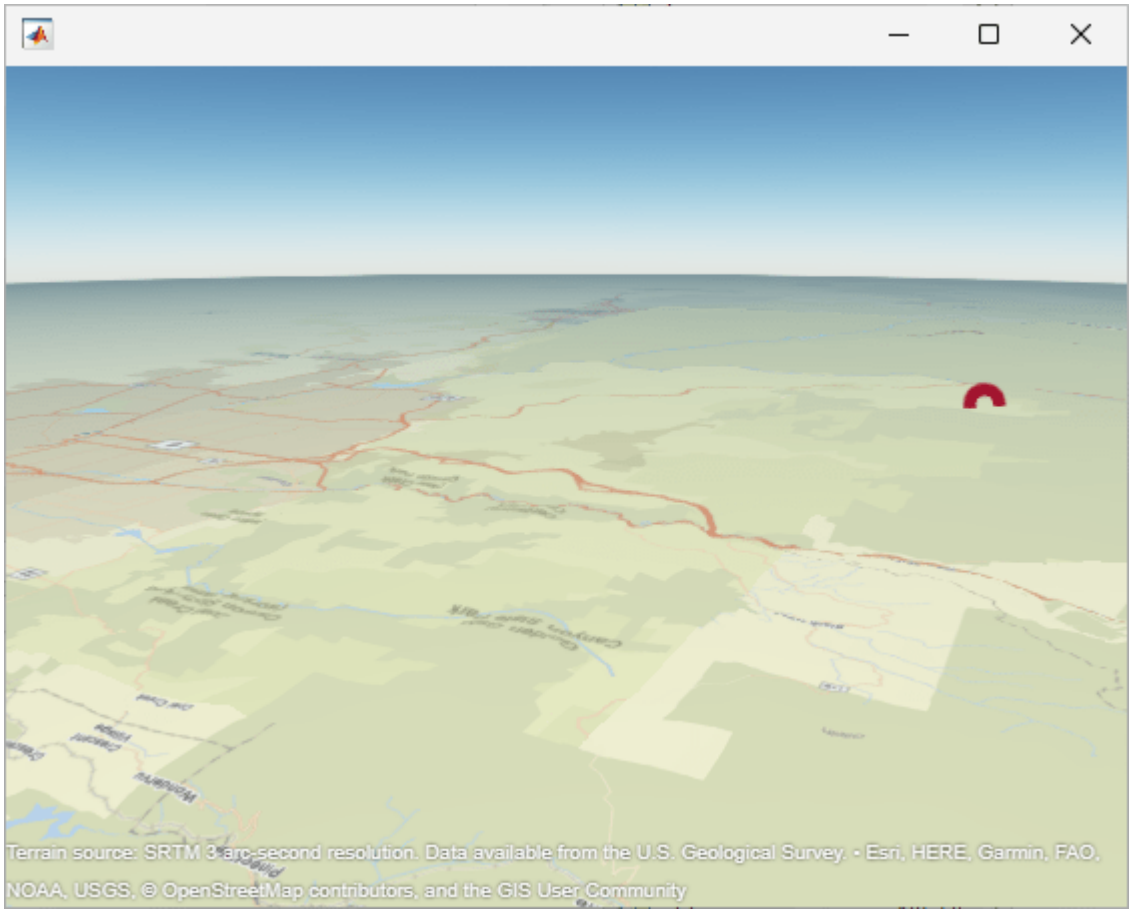
```
rdrgain = beamwidth2gain(hpbw,"CosineRectangular"); % Transmit and receive antenna gain (dB)
```

Define Location

The next section defines the radar location. The radar is mounted on a tall tower 100 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height referenced to the mean sea level (MSL). Visualize the location using `geoplot3`. The radar will appear as a red circular point in the upper right-hand corner of the image.

```
% Radar location
rdrlat = 39.5;           % Radar latitude (deg)
rdrlon = -105.5;         % Radar longitude (deg)
rdrtowerht = 100;        % Antenna height (m)
```

```
surfaceHtAtRadar = 2812; % Surface height at radar location (m)
rdralt = surfaceHtAtRadar + rdrtowerht; % Radar altitude (m)

% Import the relevant terrain data from the United States Geological
% Survey (USGS)
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.";
[Z,R] = readgeoraster(dtedfile,OutputType="double");

% Visualize the location including terrain using the geographic globe plot
addCustomTerrain("southboulder",dtedfile,Attribution=attribution);

hTerrain = uifigure;

g = geoglobe(hTerrain,Terrain="southboulder",Basemap="streets");
hold(g,"on")
h_rdrtraj = geoplot3(g,rdrlat,rdrlon,rdralt,"o", ...
    Color=[0.6350 0.0780 0.1840],LineWidth=6,MarkerSize=6);

campos(g,40.0194565714502,-105.564712896622,13117.1497971643);
camheading(g,150.8665488888147);
campitch(g,-16.023558618352187);
```



Investigate Free Space Range

For this example, the free space range `rfs` is the range at which a target has the specified signal-to-noise ratio (SNR). Assume a large target with a 10 dBsm radar cross section (RCS). At this range, the free space target SNR can be calculated as follows.

```
trcs = db2pow(10); % RCS (m^2)
lambda = freq2wavelen(fc); % Wavelength (m)
tsnr = radareqsnr(lambda,rfs,rdrppower,rdrpulsew, ...
    "RCS",trcs,"Gain",rdrgain)
```

```
tsnr = -0.7449
```

For our surveillance radar, the desired performance index is a probability of detection (Pd) of 0.8 and probability of false alarm (Pfa) below 1e-3. To make the radar system design more feasible, you can use a pulse integration technique to reduce the required SNR. For this system, assume noncoherent integration of 64 pulses. A good approximation of the minimum SNR needed for a detection at the specified Pd and Pfa can be computed by the `detectability` function as follows. Note that the free space SNR satisfies the detectability requirement.

```
Pd = 0.8;
Pfa = 1e-3;
minsnr = detectability(Pd,Pfa,64);
isdetectable = tsnr >= minsnr
```

```
isdetectable = logical
   1
```

The `toccgh` function allows you to translate receiver detection probabilities into track probabilities. Assuming the default tracker, the required Pd and Pfa translate to the following probability of target track (Pdt) and probability of false track (Pft).

```
[Pdt,Pft] = toccgh(Pd,Pfa)
```

```
Pdt = 0.9608
```

```
Pft = 1.0405e-06
```

The Pd and Pfa requirements enable a track probability of about 96% and a false track probability on the order of 1e-6.

Plot Vertical Coverage

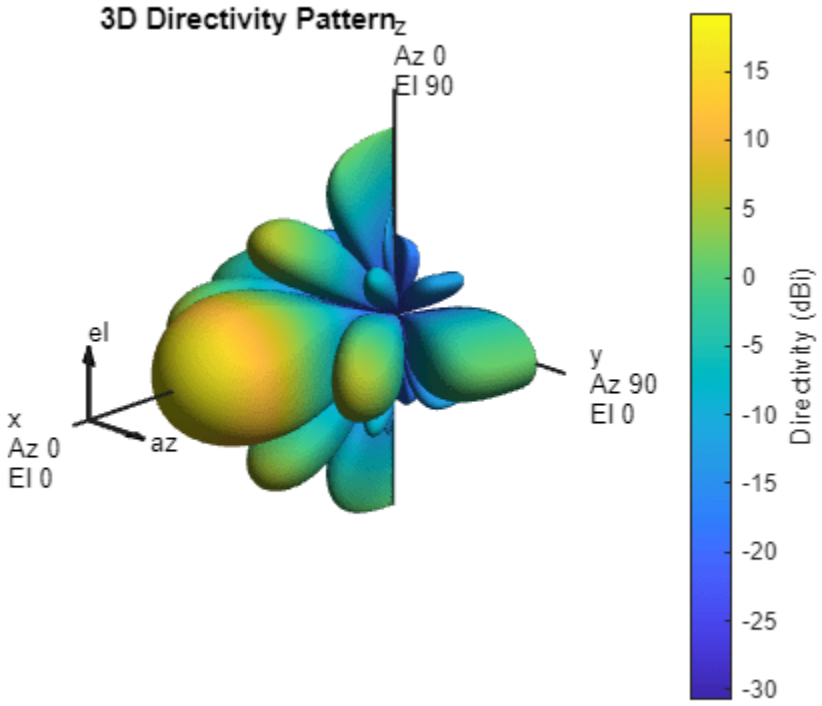
The vertical coverage pattern is visualized using a Blake chart, also known as a range-height-angle chart. The range along the *x*-axis is the propagated range and the height along the *y*-axis is relative to the origin of the ray.

The vertical coverage contour is calculated using the `radarvcd` function. The default permittivity model in `radarvcd` is based on a sea surface. Such a model is not applicable for the defined scenario, which is over land. Thus, the first step in simulating more realistic propagation is to select a more appropriate permittivity. Use the `earthSurfacePermittivity` function with the vegetation flag. Assume an ambient temperature of 21.1 degrees Celsius, which is about 70 degrees Fahrenheit. Assume a gravimetric water content of 0.3.

```
temp = 21.1; % Ambient temperature (degrees Celsius)
gwc = 0.3; % Gravimetric water content
[~,~,epsc] = earthSurfacePermittivity("vegetation",fc,temp,gwc);
```

Next, specify the antenna. Simulate the antenna as a theoretical sinc antenna pattern and plot.

```
hBeam = phased.SincAntennaElement(Beamwidth=hpbw);
pattern(hBeam,fc);
```



Get the elevation pattern at 0 degrees azimuth by using the `helperGetVoltagePattern` on page 11-75 function.

```
elAng = -90:0.01:90;
pat = helperGetVoltagePattern(hBeam,fc,0,elAng);
```

Specify the atmosphere and calculate the corresponding effective Earth radius. Since the latitude of the radar in this example is 39.5 degrees, a mid-latitude atmosphere model would be most appropriate. Assume that the time period is during the summer.

```
% Set effective Earth radius using the mid latitude atmosphere model
% in summer
[nidx,N] = refractiveidx([0 1e3],LatitudeModel="Mid",Season="Summer");
RGradient = (nidx(2) - nidx(1))/1e3;
Re = effearthradius(RGradient); % m
```

Next, calculate the vertical coverage pattern using the `radarvcd` function.

```
[vcpkm,vcpang] = radarvcd(fc,rfs.*1e-3,rdrtowerht, ...
    SurfaceRelativePermittivity=epsc, ...
    SurfaceHeightStandardDeviation=30, ...
    Vegetation="Trees", ...
    AntennaPattern=pat,PatternAngles=elAng, ...
```

```
    TiltAngle=tiltAng, ...
    EffectiveEarthRadius=Re, ...
    MinElevation=minEl, ...
    ElevationStepSize=elStepSize, ...
    MaxElevation=maxEl);
```

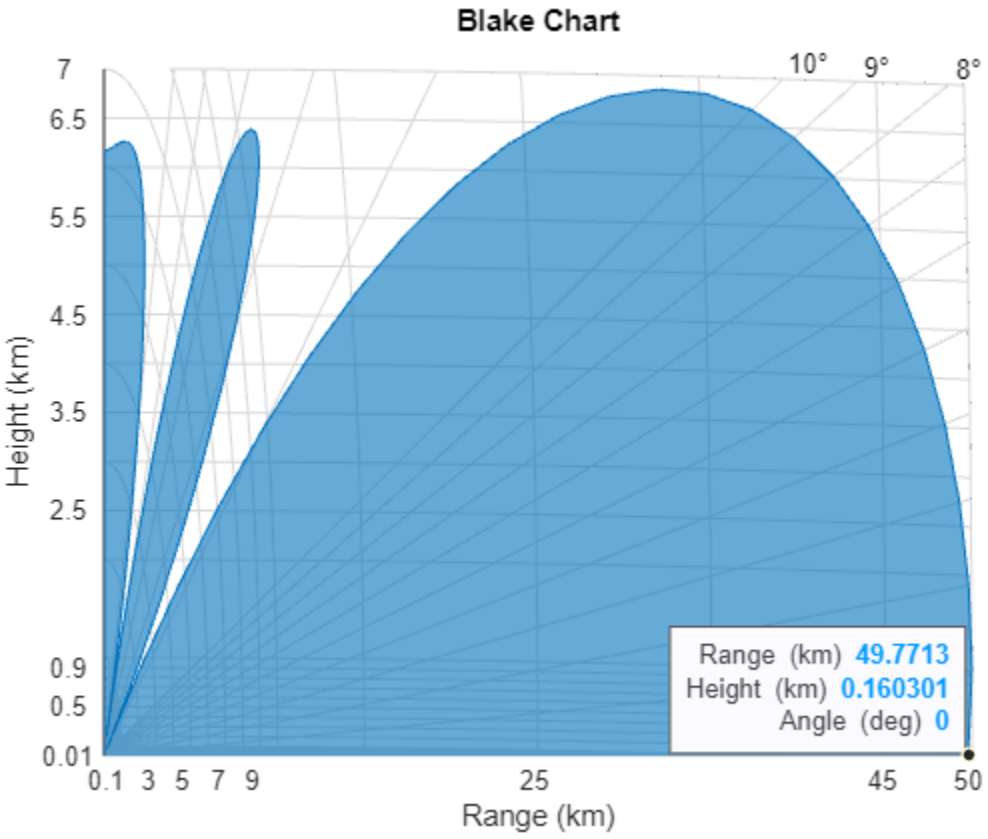
Use the surface height at the site to obtain the surface refractivity from `refractiveidx`.

```
% Calculate an appropriate surface
[~,Ns] = refractiveidx(surfaceHtAtRadar,LatitudeModel="Mid",Season="Summer");
```

Plot the vertical coverage using the `blakechart` function. The `blakechart` axes are formed using the Central Radio Propagation Laboratory (CRPL) reference atmosphere. The CRPL model is widely used and models the refractivity profile as an exponential decay versus height, which has been shown to be an excellent approximation for a normal atmosphere (an atmosphere that is not undergoing anomalous propagation like ducting). The CRPL model is tailored by setting the surface height refractivity and the refraction exponent to a value that is appropriate for the location in which the system is operating and the time of year.

Update the surface refractivity and the refraction exponent inputs in the function call. In the subsequent plot, the constant-signal-level of the contour is given by the previously calculated target SNR.

```
rexp = refractionexp(Ns);
blakechart(vcpkm,vcpang,ScalePower=1,SurfaceRefractivity=Ns,RefractionExponent=rexp)
```



Visualize 3-D Vertical Coverage over Terrain

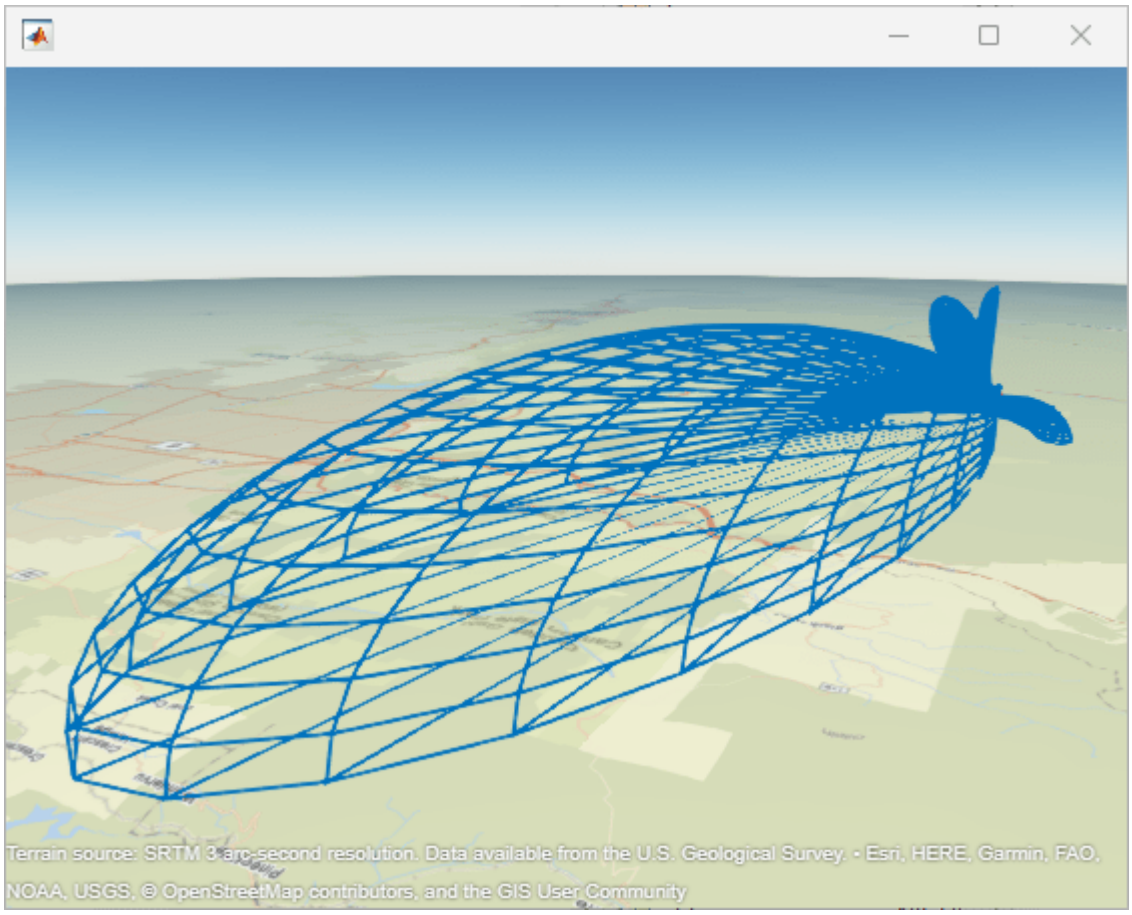
The next section calculates the vertical coverage at the specified azimuth intervals. The vertical coverage range and angle is converted to height using the `range2height` function using the `CRPL` method. The range-height-angle values are then converted to Cartesian.

```
% Initialize Cartesian X, Y, Z outputs
numAz = numel(azAng);
numRows = numel(minEl:elStepSize:maxEl)+1;
vcpX = zeros(numRows,numAz);
vcpY = zeros(numRows,numAz);
vcpZ = zeros(numRows,numAz);
wgs84 = wgs84Ellipsoid;

% Obtain vertical coverage contour in Cartesian coordinates
for idx = 1:numAz
    % Get elevation pattern at this azimuth
    pat = helperGetVoltagePattern(hBeam,fc,azAng(idx),elAng);

    % Get terrain height information
    [xdir,ydir,zdir] = sph2cart(deg2rad(azAng(idx)+azRotation),0,1e3);
    [xlat,ylon,zlon] = enu2geodetic(xdir,ydir,zdir,rdrlat,rdrlon,rdralt,wgs84);
    [~,~,~,htsurf] = los2(Z,R,rdrlat,rdrlon, ...
        xlat,ylon,rdralt,zlon,"MSL","MSL");
    htstdSurf = std(htsurf(~isnan(htsurf)));

    % Calculate vertical coverage pattern
    [vcp,vcpang] = radarvcd(fc,rfs,rdrtowerht, ...
        RangeUnit="m", ...
        HeightUnit="m", ...
        SurfaceRelativePermittivity=epsc, ...
        SurfaceHeightStandardDeviation=htstdSurf, ...
        Vegetation="Trees", ...
        AntennaPattern=pat,PatternAngles=elAng, ...
        TiltAngle=tiltAng, ...
        EffectiveEarthRadius=Re, ...
        MinElevation=1, ...
        ElevationStepSize=1, ...
        MaxElevation=90);

    % Calculate associated heights
    vcpht = range2height(vcp,rdrtowerht,vcpang, ...
        Method="CRPL",MaxNumIterations=2,Tolerance=1e-2, ...
        SurfaceRefractivity=Ns,RefractionExponent=rexp);

    % Calculate true slant range and elevation angle to target
    [~,vcpgeomrt,vcpelt] = ...
        height2range(vcpht,rdrtowerht,vcpang, ...
        Method="CRPL", ...
        SurfaceRefractivity=Ns,RefractionExponent=rexp);

    % Set values for this iteration
    vcpelt = [0;vcpelt(:);vcpang(end)];
    vcpgeomrt = [0;vcpgeomrt(:);0]; % km

    % Convert range, angle, height to Cartesian X, Y, Z
    [vcpX(:,idx),vcpY(:,idx),vcpZ(:,idx)] = sph2cart(...
        deg2rad(azAng(idx).*ones(numRows,1)+azRotation), ...
```

```
        deg2rad(vcpelt),vcpgeomrt);
end
```

Convert the Cartesian vertical coverage values to geodetic and plot on the globe with the help of the function `helperCreateMesh` on page 11-75. The three-dimensional vertical coverage appears as a blue mesh.

```
[vcpLat,vcpLon,vcpAlt] = enu2geodetic(vcpX,vcpY,vcpZ,rdrlat,rdrlon,rdralt,wgs84);
% Organize data into a mesh for plotting
[xMesh,yMesh,zMesh] = helperCreateMesh(vcpLat,vcpLon,vcpAlt);
geoplot3(g,xMesh,yMesh,zMesh,LineWidth=1,Color=[0 0.4470 0.7410]);
```



Summary

This example discussed a method to calculate a three-dimensional vertical coverage, discussing ways to tailor the analysis to the local atmospheric conditions. The visualization created gives insight into the vertical performance of the radar system, considering the interference from the surface reflections and refraction.

Helper Functions

```
% Clean up by closing the geographic globe and removing the imported
% terrain data.
if isvalid(hTerrain)
    close(hTerrain)
```

```
end
removeCustomTerrain("southboulder")


function pat = helperGetVoltagePattern(hBeam,fc,azAng,elAng)
% Obtain voltage pattern from antenna element
numAz = numel(azAng);
numEl = numel(elAng);
pat = zeros(numAz,numEl);
for ia = 1:numAz
    for ie = 1:numEl
        pat(ia,ie) = step(hBeam,fc,[azAng(ia);elAng(ie)]);
    end
end
end

function [xMesh,yMesh,zMesh] = helperCreateMesh(x,y,z)
% Organizes the inputs into a mesh, which can be plotted using geoplot3
numPts = numel(x);
xMesh = zeros(2*numPts,1);
yMesh = zeros(2*numPts,1);
zMesh = zeros(2*numPts,1);
[nrows,ncols] = size(x);
idxStart = 1;
idxEnd = nrows;
for ii = 1:ncols
    if mod(ii,2) == 0
        xMesh(idxStart:idxEnd) = x(:,ii);
        yMesh(idxStart:idxEnd) = y(:,ii);
        zMesh(idxStart:idxEnd) = z(:,ii);
    else
        xMesh(idxStart:idxEnd) = flipud(x(:,ii));
        yMesh(idxStart:idxEnd) = flipud(y(:,ii));
        zMesh(idxStart:idxEnd) = flipud(z(:,ii));
    end

    idxStart = idxEnd + 1;
    if ii ~= ncols
        idxEnd = idxStart + nrows -1;
    else
        idxEnd = idxStart + ncols -1;
    end
end

for ii = 1:nrows
    if mod(ii,2) == 0
        xMesh(idxStart:idxEnd) = x(ii,:);
        yMesh(idxStart:idxEnd) = y(ii,:);
        zMesh(idxStart:idxEnd) = z(ii,:);
    else
        xMesh(idxStart:idxEnd) = fliplr(x(ii,:));
        yMesh(idxStart:idxEnd) = fliplr(y(ii,:));
        zMesh(idxStart:idxEnd) = fliplr(z(ii,:));
    end
    idxStart = idxEnd + 1;
    idxEnd = idxStart + ncols -1;
```

```
end
end
```

References

[1] Barton, D. K. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.

[2] Bean, B. R., and G. D. Thayer. "Central Radio Propagation Laboratory Exponential Reference Atmosphere." Journal of Research of the National Bureau of Standards, Section D: Radio Propagation 63D, no. 3 (November 1959): 315. https://doi.org/10.6028/jres.063D.031.

[3] Blake, L. V. "Radio Ray (Radar) Range-Height-Angle Charts." Naval Research Laboratory, January 22, 1968.

[4] Blake, L. V. "Ray Height Computation for a Continuous Nonlinear Atmospheric Refractive-Index Profile." Radio Science 3, no. 1 (January 1968): 85-92. https://doi.org/10.1002/rds19683185.

See Also

Functions

`addCustomTerrain` | `geoglobe` | `geoplot3` | `radarvcd` | `blakechart`

Related Examples

- “Radar Performance Analysis over Terrain” on page 11-58

Process Large GeoTIFF File as Blocked Image

This example shows how to efficiently apply and visualize image processing operations on a large GeoTIFF file.

The goal of this example is to detect boats on an image of a river. Applying the boat detection algorithm directly to a large file requires a lot of memory and potentially wastes computational resources. The example mitigates these challenges by working with the GeoTIFF file as a blocked image. The blocked image approach can scale to very large files because it loads and processes one block of data at a time. The blocked image approach is also computationally efficient because it can omit blocks of nonmeaningful data, such as regions of land, for the boat detection algorithm.

The steps to apply the boat detection algorithm to a blocked image include:

- Read the GeoTIFF file into a `blockedImage` object.
- Create a low-resolution version of the image.
- Create a mask of the water regions of interest (ROIs) at the low resolution level.
- Apply the boat detection algorithm, at the high resolution level, to blocks that overlap with the mask.

Create Multiresolution Blocked Image

Create a `blockedImage` object that reads the GeoTIFF file. This image is of Boston and the surrounding region, including the Charles River and Boston Harbor. The image has one resolution level.

```
bim = blockedImage("boston.tif");
```

Create an adapter that reads TIFF files and applies JPEG compression.

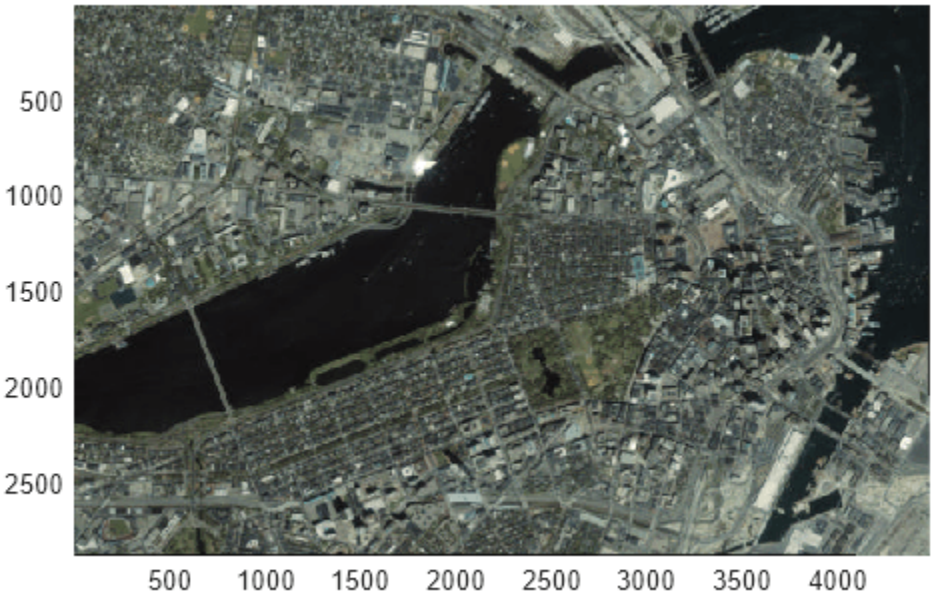
```
adapter = images.blocked.TIFF;
adapter.Compression = Tiff.Compression.JPEG;
```

Create a two-level `blockedImage` object by using the `makeMultiLevel2D` function. Specify the block size as 1024-by-1024 pixels, which is generally appropriate for various image sizes and machine configurations. Specify the scale factors as the two-element vector `[1 1/8]`. The image at the first resolution level consists of the original large image in `bim`. The image at the second resolution level is 1/8 the size of the original image in each dimension.

```
timestamp = datetime("now",Format="yyyy-MM-dd-HH-mm-ss");
outputFile = "boston_multiLevel_" + string(timestamp) + ".tif";
bim2 = makeMultiLevel2D(bim,BlockSize=[1024 1024],Scales=[1 1/8], ...
    OutputLocation=outputFile,Adapter=adapter);
```

Display the multiresolution blocked image using the `bigimageshow` function. As you zoom in and out of the displayed image, the `bigimageshow` function switches between the resolution levels.

```
bigimageshow(bim2)
```



Create Mask of Water ROI

You can process large images more quickly by applying the operations to only regions of interest (ROIs). For this example, because the ROIs consist of bodies of water, you want the boat detection algorithm to run on only regions of water and not on regions of land.

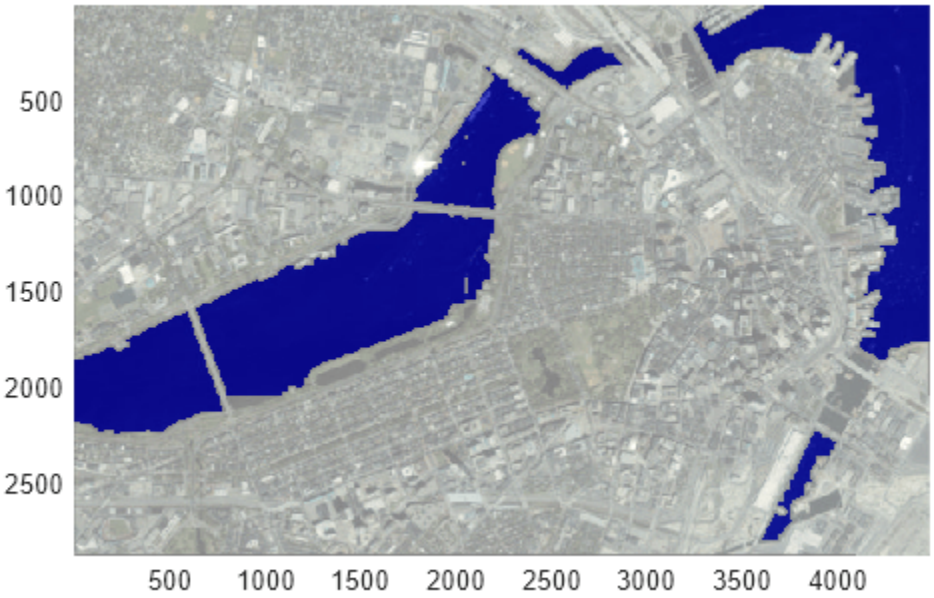
For this image, regions of water are darker than regions of land. Therefore, you can identify the water ROIs by applying a threshold to the pixel values. Alternatively, if you have a shapefile containing geological or political boundaries, you can create a mask from the shapefile. For an example, see “Crop and Mask Large GeoTIFF File Using Shapefile” on page 11-85.

Specify a pixel value threshold of `50`. Then, apply the threshold to the image at the coarse resolution level and clean up the resulting mask by using the `detectWater` on page 11-82 helper function, which is defined at the end of the example. Call the `detectWater` helper function on all blocks in the blocked image by using the `apply` function. Ensure that the first block is full by specifying a block size of 256-by-256 pixels.

```
maxWaterPixelValue = 50;
lvl = bim2.NumLevels;
waterMask = apply(bim2,@(bs)detectWater(bs,maxWaterPixelValue), ...
    BlockSize=[256 256],Level=lvl);
```

Display the mask in blue over a grayscale version of the original image.

```
hb = bigimageshow(bim2);
showlabels(hb,waterMask,Colormap=[1 1 1; 0 0 1])
```



Detect Boats Within ROI

Identify the blocks at the high resolution level that are within the ROI by using the `selectBlockLocations` function and specifying the `Masks` name-value argument. To include all blocks with at least one pixel within the ROI, also specify the `InclusionThreshold` name-value argument as `0`.

```
blockSet = selectBlockLocations(bim,BlockSize=[1024 1024], ...
    Masks=waterMask,InclusionThreshold=0);
```

Specify the minimum pixel values and areas for the boat detection algorithm. For the pixel values, use a minimum threshold of `150`, which corresponds to bright pixels. For the areas, specify a minimum area of `5` pixels.

```
minBoatPixelValue = 150;
minBoatArea = 5;
```

Apply the boat detection algorithm by using the `detectBoats` on page 11-82 helper function, which is defined at the end of the example. Call the `detectBoats` helper function on all blocks of the high resolution image by using the `apply` function. To limit processing to the identified blocks within the ROI, specify the `BlockLocationSet` name-value argument. To improve the accuracy of detecting boats that span two blocks, specify the `BorderSize` name-value argument.

```
detections = apply(bim, ...
    @(block,waterMask)detectBoats(block,waterMask,minBoatPixelValue,minBoatArea), ...
    BlockLocationSet=blockSet,BorderSize=[10 10],ExtraImages=waterMask);
```

Load all detections from the resulting `blockedImage` output into an array.

```
detections = gather(detections);
centroidsXY = vertcat(detections.centroidsXY);
```

GeoTIFF files have raster reference information embedded in the metadata. This information is required to convert the coordinates of a detected object from raster coordinates to map coordinates.

Get the raster reference object for the image by using the `georasterinfo` function, then convert the raster coordinates to map coordinates.

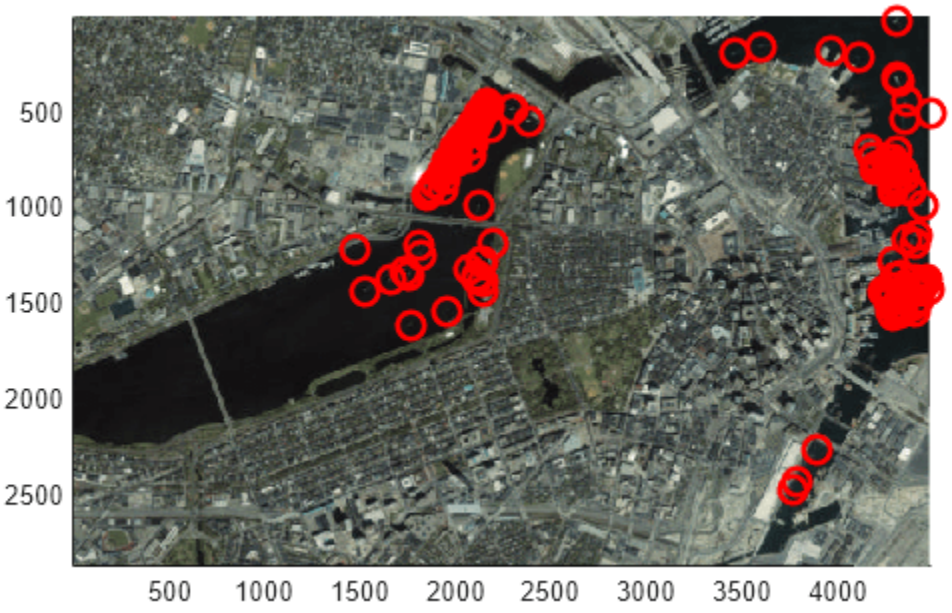
```
ginfo = georasterinfo(bim.Source);
R = ginfo.RasterReference;
[centroidsWorldX,centroidsWorldY] = intrinsicToWorld(R,centroidsXY(:,1),centroidsXY(:,2));
centroidsWorld = [centroidsWorldX centroidsWorldY];
```

Display Results

Raster Coordinates

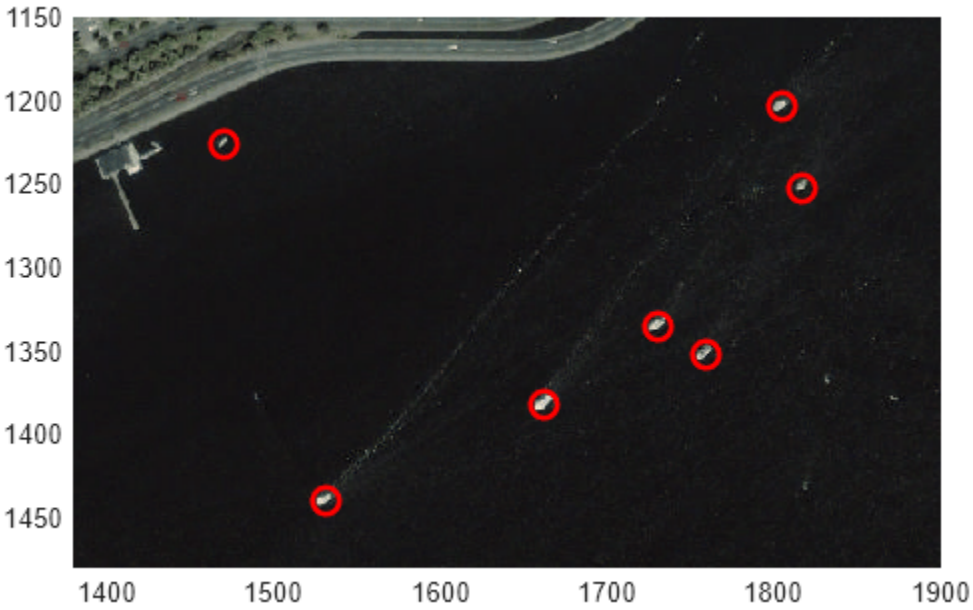
Display the results in raster coordinates by using the `bigimageshow` function. Plot the centroids of the detected boats over the image.

```
figure
bigimageshow(bim2);
hold on
plot(centroidsXY(:,1),centroidsXY(:,2),"ro",MarkerSize=10,LineWidth=2)
```



When you zoom in, `bigimageshow` switches to higher available resolutions.

```
xlim([1380 1900])
ylim([1150 1480])
```



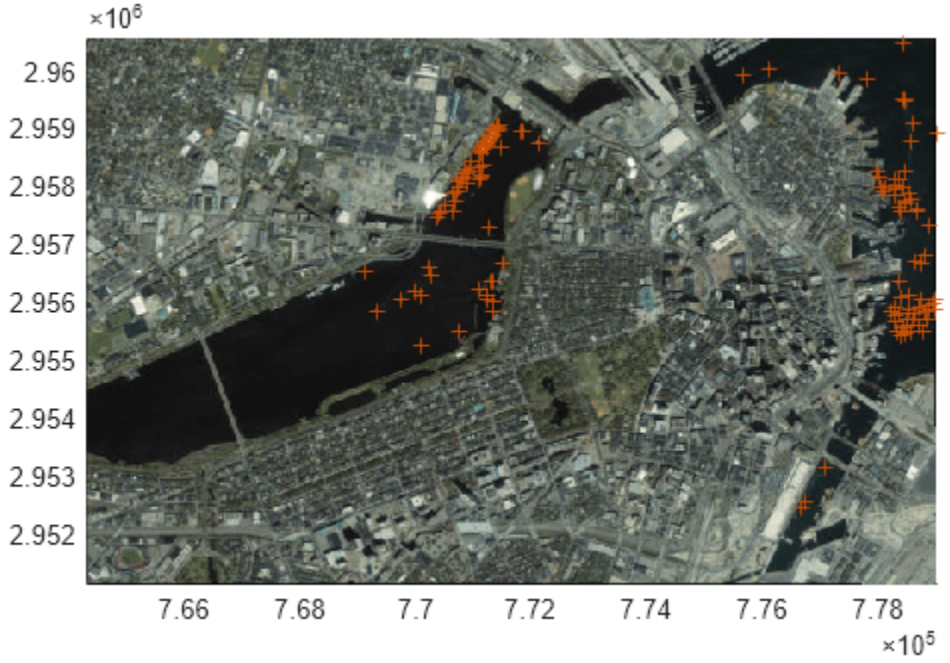
Map Coordinates

To display the results in map coordinates, use the low-resolution image from the blocked image. Update the corresponding raster reference object to match the size of this level.

```
imLowRes = gather(bim2);
RLowRes = R;
RLowRes.RasterSize = size(imLowRes);
```

Display the image and detections in map coordinates by using the `mapshow` function.

```
figure
mapshow(imLowRes,RLowRes)
mapshow(centroidsWorld(:,1),centroidsWorld(:,2),DisplayType="point")
```



Helper Functions

The `detectWater` helper function detects large bodies of water in a blocked image. Within the function:

- Identify dark pixels, which likely correspond to water, by using intensity thresholding. ROIs consist of pixels with a value less than the `maxWaterPixelValue` argument.
- Clean up the ROIs by filling holes using the `imfill` function and by shrinking the ROI using the `imerode` function.
- Retain only large ROIs that correspond to bodies of water by using the `bwareafilt` function. This example defines a large ROI as one containing at least 500 pixels.

```
function bw = detectWater(bs,maxWaterPixelValue)
    bw = bs.Data(:,:,1) < maxWaterPixelValue;
    bw = imfill(bw,"holes");
    bw = imerode(bw,ones(3));
    bw = bwareafilt(bw,[500 Inf]);
end
```

The `detectBoats` helper function detects boats on bodies of water using intensity thresholding. Within the function:

- Set all block pixels outside the mask of water ROIs to 0.
- Identify bright pixels, which likely correspond to boats, by using intensity thresholding. Detections consist of pixels with a value greater than the `minBoatPixelValue` argument.
- Measure the area and centroid of all detections using the `regionprops` function.

- Remove detections with an area smaller than the `minBoatArea` argument.
- Remove detections on the border by using the `clearBorderData` on page 11-83 helper function, which is defined at the end of this example.
- Return the centroids of the detections.

```
function detections = detectBoats(block,waterMaskBlock, ...
    minBoatPixelValue,minBoatArea)

    % Resize the mask block to match the size of the image block
    waterMaskBlock = imresize(waterMaskBlock,size(block.Data,[1 2]),"nearest");

    % Set pixels outside the ROI to 0 and detect objects using intensity
    % thresholding
    imMasked = im2gray(block.Data);
    imMasked(~waterMaskBlock) = 0;
    bw = imMasked > minBoatPixelValue;

    % Measure the area and centroid of detected objects
    stats = regionprops(bw,["Area" "Centroid"]);

    % Remove detections with small area
    stats([stats.Area] < minBoatArea) = [];

    % Remove detections on the border
    stats = clearBorderData(stats,block);

    if isempty(stats)
        detections.centroidsXY = [];
    else
        blockCentroidsXY = vertcat(stats.Centroid);
        % Convert coordinates from row-column to X-Y order
        blockOffsetXY = fliplr(block.Start(1:2));
        detections.centroidsXY = blockCentroidsXY + blockOffsetXY;
    end
end
```

The `clearBorderData` helper function removes detections whose centroids are in the border area of a block.

```
function stats = clearBorderData(stats,block)

    % Convert coordinates from row-column to X-Y order
    lowEdgeXY = fliplr(block.BorderSize(1:2) + 1);
    highEdgeXY = fliplr(block.BlockSize(1:2) - block.BorderSize(1:2));

    % Find indices of centroids that are inside the current block, excluding
    % the border region
    centroidsXY = reshape([stats.Centroid],2,[]);
    centroidsXY = round(centroidsXY);
    inBlock = centroidsXY(1,:)>=lowEdgeXY(1) & centroidsXY(1,:)<=highEdgeXY(1) ...
        & centroidsXY(2,:)>=lowEdgeXY(2) & centroidsXY(2,:)<=highEdgeXY(2);

    % Filter stats to retain only inside block detections
```

```
    stats = stats(inBlock);
end
```

See Also

Functions
`makeMultiLevel2D` | `selectBlockLocations` | `regionprops` | `bigimageshow` | `apply`

Objects
`blockedImage`

Related Examples

- “Process Blocked Images Efficiently Using Mask” (Image Processing Toolbox)
- “Crop and Mask Large GeoTIFF File Using Shapefile” on page 11-85

Crop and Mask Large GeoTIFF File Using Shapefile

This example shows how to crop, mask, and export a large GeoTIFF file using a shape from a shapefile.

The goals of this example are to crop a large GeoTIFF file to the smallest extent that encompasses the shape in the shapefile, and to highlight the image data within the shape by applying a mask that sets the value of all pixels outside the shape to `0`. Applying the cropping and masking operations directly to a large file requires a lot of memory and potentially wastes computational resources. The example mitigates these challenges by working with the GeoTIFF file as a blocked image. The blocked image approach can scale to very large files because it loads and processes one block of data at a time. The blocked image approach is also computationally efficient because it can omit blocks of nonmeaningful data, such as regions outside the shape.

The steps to process a large GeoTIFF file with cropping and masking operations include:

- Read the GeoTIFF file into a `blockedImage` object.
- Load the shape from the shapefile.
- Calculate the bounding box of the shape in raster coordinates, and crop the high-resolution image to the bounding box.
- Convert the shape from a set of polygon coordinates to a low-resolution raster mask.
- Mask blocks of the cropped high-resolution image that overlap with the low-resolution mask of the shape.
- Write the processed data to a TIFF file, and insert the required GeoTIFF tags to convert the file to a GeoTIFF file.

Create Multiresolution Blocked Image

Create a `blockedImage` object that reads the GeoTIFF file. This image is of Boston and the surrounding region. The image has one resolution level.

```
bim = blockedImage("boston.tif");
```

Create an adapter that reads TIFF files and applies JPEG compression.

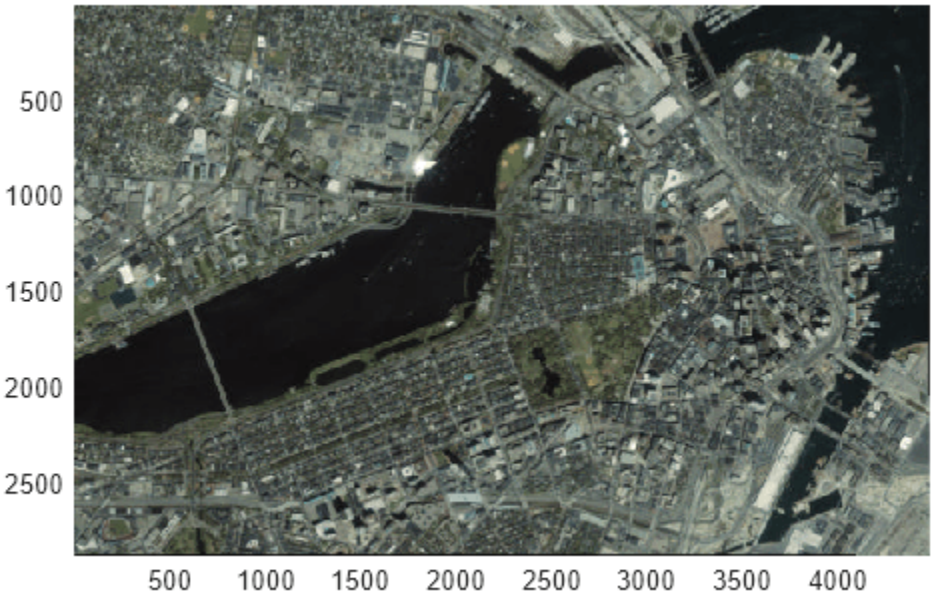
```
adapter = images.blocked.TIFF;
adapter.Compression = Tiff.Compression.JPEG;
```

Create a two-level `blockedImage` object by using the `makeMultiLevel2D` function. Specify the block size 1024-by-1024 pixels, which is generally appropriate for various image sizes and machine configurations. Specify the scale factors as the two-element vector `[1 1/8]`. The image at the first resolution level consists of the original large image in `bim`. The image at the second resolution level is 1/8 the size of the original image in each dimension.

```
timestamp = datetime("now",Format="yyyy-MM-dd-HH-mm-ss");
outputFile = "boston_multiLevel_" + string(timestamp) + ".tif";
bim2 = makeMultiLevel2D(bim,BlockSize=[1024 1024],Scales=[1 1/8], ...
    OutputLocation=outputFile,Adapter=adapter);
```

Display the multiresolution blocked image in raster coordinates by using the `bigimageshow` function. As you zoom in and out of the displayed image, the `bigimageshow` function switches between the resolution levels.

```
bigimageshow(bim2)
```



GeoTIFF files have raster reference information embedded in the metadata. This information is required to transform the detection of an object in pixel coordinates to map coordinates. Get the raster reference object for the high-resolution image using the `georasterinfo` function.

```
ginfo = georasterinfo(bim.Source);
R = ginfo.RasterReference;
```

Load Shapefile

Read the shape from the shapefile by using the `readgeotable` function. The shapefile contains a hand drawn outline of two parks in the city of Boston. The shape uses the same projected coordinate reference system (CRS) as the image.

```
shptbl = readgeotable("bostonparks.shp");
```

To display the shape in map coordinates, load the low-resolution image from the blocked image. Update the corresponding raster reference object to match the size of this level.

```
imLowRes = gather(bim2);
RLowRes = R;
RLowRes.RasterSize = size(imLowRes);
```

Display the shape over the low-resolution image in map coordinates by using the `mapshow` function.

```
figure
mapshow(imLowRes,RLowRes)
mapshow(shptbl,EdgeColor="y",FaceColor="none")
```

Crop Image to Extents of Shape

Crop the image to the smallest extent that encompasses the shape.

Convert the coordinates of the shape from map coordinates to raster coordinates by using the raster reference object.

```
tbl = geotable2table(shptbl,["XWorld" "YWorld"]);
xWorld = tbl.XWorld{1};
yWorld = tbl.YWorld{1};
[x,y] = worldToIntrinsic(R,xWorld,yWorld);
parkPoly = [x(:) y(:)];
```

Find the bounding box of the shape in raster coordinates.

```
[xmin,xmax] = bounds(parkPoly(:,1));
[ymin,ymax] = bounds(parkPoly(:,2));
cropStart = floor([ymin xmin]);
cropStart = max(cropStart,1);
cropEnd = ceil([ymax xmax]);
cropEnd = min(cropEnd,bim2.Size(1,1:2));
```

Crop the blocked image to the bounding box using the `crop` function. The `crop` function updates the blocked image metadata with the new extents of the image at each resolution level.

```
bimCrop = crop(bim2,cropStart,cropEnd);
```

Update the corresponding reference object to use the extents of the cropped image.

```
rowLimits = [cropStart(1) cropEnd(1)];
colLimits = [cropStart(2) cropEnd(2)];
Rcrop = cropToBlock(R,rowLimits,colLimits);
```

To verify the results of the cropping operation, display the shape as a polygon over the cropped image. The `bigimageshow` and `drawpolygon` functions display the cropped blocked image and the shape polygon using raster coordinates.

```
bigimageshow(bimCrop)
hp = drawpolygon(Position=parkPoly);
```

Create Raster Mask

You can process large images more quickly by applying the operations to only regions of interest (ROIs). For this example, the ROI consists of pixels within the shape polygon. Convert the polygon vertices to a raster mask at the coarse resolution level by using the `poly2BlockedImage` function. Ensure that the first block is full by specifying a block size of 256-by-256 pixels.

```
lvl = bim2.NumLevels;
maskSize = bim2.Size(lvl,[1 2]);
mask = polyToBlockedImage({parkPoly},true,maskSize,BlockSize=[256 256], ...
    WorldStart=bimCrop.WorldStart(1,:),WorldEnd=bimCrop.WorldEnd(1,:));
```

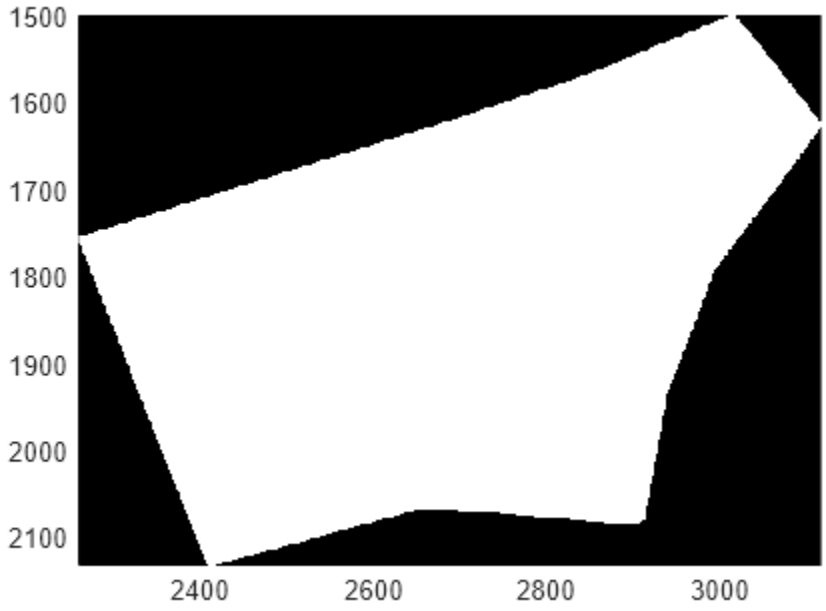
This example performs the optional step of dilating the mask. Dilation extends the masked region slightly beyond the initial shape boundary. This step can improve the result of masking the image when you know the shape boundary to be slightly inaccurate around small boundary features, or you need to include additional image details just past the initial boundary.

Dilate the mask at the coarse resolution level by using the `imdilate` function. Call the `imdilate` function on all blocks in the mask by using the `apply` function.

```
mask = apply(mask,@(block)imdilate(block.Data,ones(3)));
```

Display the mask in raster coordinates by using the `bigimageshow` function.

```
bigimageshow(mask)
```



Apply Raster Mask to Cropped Image

Identify the image blocks at the high resolution level that are within the ROI by using the `selectBlockLocations` function and specifying the `Masks` name-value argument. To include all image blocks with at least one pixel within the ROI, also specify the `InclusionThreshold` name-value argument as `0`. Ensure that the first block is full by specifying a block size of 256-by-256 pixels.

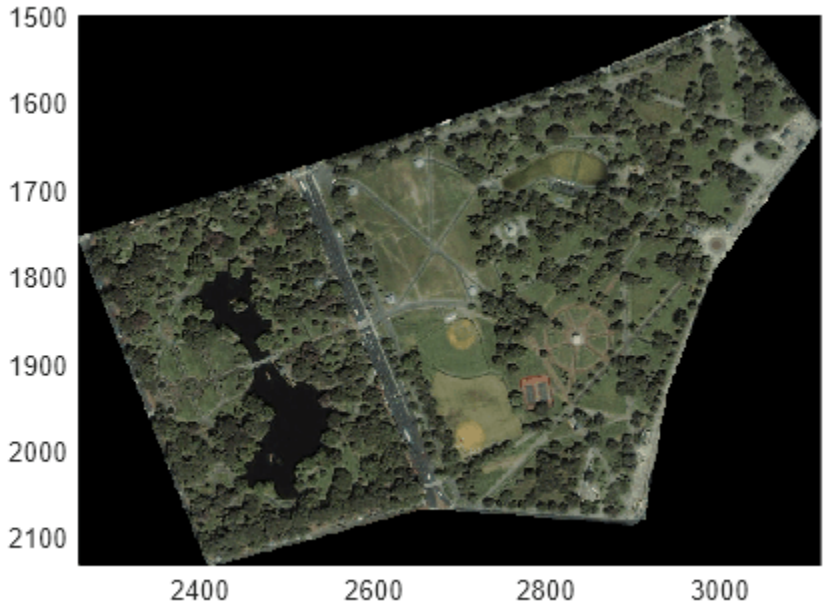
```
bls = selectBlockLocations(bimCrop,BlockSize=[256 256], ...
    Masks=mask,InclusionThreshold=0);
```

Apply the mask to the image at the finest resolution level by using the `applyMask` on page 11-91 helper function, which is defined at the end of the example. Call the `applyMask` helper function on all blocks of the high-resolution image by using the `apply` function. To limit processing to the identified blocks within the ROI, specify the `BlockLocationSet` name-value argument. Note that the `apply` function automatically sets all image blocks outside the ROI to the default value of `0`. To write the processed blocks to a TIFF file, specify the `OutputLocation` name-value argument with the target output file, and specify the `Adapter` name-value argument as a TIFF adapter.

```
outputFileMasked = "bostonparks_" + string(timestamp) + ".tif";
bimMasked = apply(bimCrop,@applyMask,ExtraImages=mask, ...
    blockLocationSet=bls,OutputLocation=outputFileMasked, ...
    Adapter=images.blocked.TIFF);
```

Display the mask in raster coordinates by using the `bigimageshow` function.

```
bigimageshow(bimMasked)
```



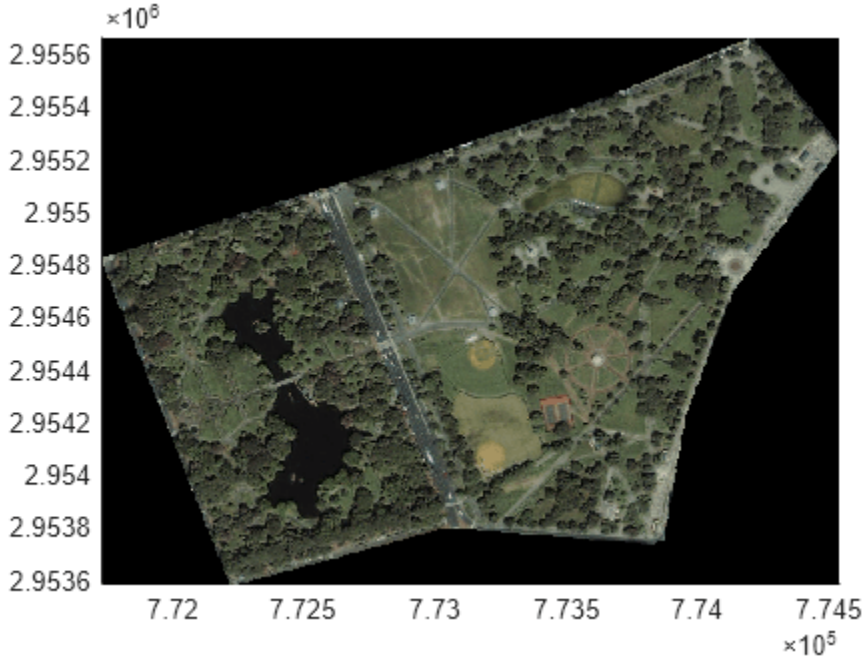
Convert TIFF File to GeoTIFF File

Convert the cropped and masked TIFF file to a valid GeoTIFF file by using the `insertCroppedGeoTIFFTags` helper function, which is attached to the example as a supporting file. The helper function generates and inserts the required GeoTIFF tags into the TIFF file. The data for the tags consists of coordinate information and additional metadata from the original GeoTIFF file.

```
insertCroppedGeoTIFFTags(outputFileMasked,"boston.tif",Rcrop)
```

To ensure that the cropped GeoTIFF file uses the correct map coordinates from the original GeoTIFF file, display the image from the cropped GeoTIFF file by using the `mapshow` function.

```
[bostonParks,RParks] = readgeoraster(outputFileMasked);
mapshow(bostonParks,RParks)
```



Helper Function

The `applyMask` helper function resizes a block of the mask to match the size of a block of image data, and sets the image data to `0` wherever the corresponding mask value is `0`.

```
function imMasked = applyMask(data,mask)
    im = data.Data;
    mask = imresize(mask,size(im,[1 2]));
    mask = cast(mask,"like",im);
    imMasked = im.*mask;
end
```

See Also

Functions
`makeMultiLevel2D` | `selectBlockLocations` | `bigimageshow` | `apply`

Objects
`blockedImage`

Related Examples

- “Process Blocked Images Efficiently Using Mask” (Image Processing Toolbox)
- “Process Large GeoTIFF File as Blocked Image” on page 11-77

Hurricane Natural Catastrophe Risk Estimation

This example measures natural catastrophe (NatCat) risk for hurricanes in an area of interest (AOI) by using simulations from a hazard model, data about property assets, and a damage function.

Hurricanes are a common natural catastrophe and a key focus for NatCat modeling in the insurance industry. Traditionally, NatCat models measure financial risk by combining three inputs [7 on page 11-107]:

- Data related to a hazard of interest. For this example, the hazard data comes from hurricane simulations [2 on page 11-107] generated from the approach used in [1 on page 11-107].
- Data related to a set of exposures. For this example, the exposure data includes the addresses of several property assets, the geographic locations of the property assets, and the median values of surrounding property assets.
- Information related to asset vulnerability. Asset vulnerability is typically calculated using a damage function, which estimates damage by combining an intensity metric with data about the asset. For this example, the intensity metric is wind speed.

NatCat risk models typically cover a short time period, often a year. A related risk model, physical climate risk, measures NatCat risk using multiple long-term climate scenarios, then models the impact of the long-term risk on financial variables such as house prices and insurance costs. Physical risk models typically cover longer time periods, often decades. As a result, NatCat risk models are useful in isolation and for physical risk models.

Within the example, you perform these tasks:

- Import, process, and visualize the hurricane simulations.
- Specify the exposure data using the addresses of properties in Florida.
- Define a damage function.
- Estimate property loss using the hurricane simulations, the exposure data, and the damage function.
- Analyze the property loss for individual properties and for a portfolio of properties.

Import and Process Hurricane Data

Import and process a file that contains simulated hurricane data [2 on page 11-107] generated from the approach used in [1 on page 11-107]. The data represents 1000 possible scenarios in the North Atlantic basin for the year 2021, where each simulated year contains zero to multiple hurricanes.

Create a temporary folder to store a text file containing hurricane data. Download the file into the folder by using the `downloadHurricaneData` helper function. The helper function is attached to the example as a supporting file.

```
dataDir = fullfile(tempdir,"HurricaneData");
if ~exist(dataDir,"dir")
    mkdir(dataDir)
end
downloadHurricaneData(dataDir)
```

Specify the name of the text file. Then, read the data into the workspace as a table. View the top rows of the table.

```
filename = fullfile(dataDir,"HurricaneData.txt");
hcanes = readtable(filename);
head(hcanes)
```

Landfall	Latitude	Longitude	MaximumWindSpeed	MinimumPressure	RadiusToMaximumW
0	8.9	314.8	20.372	993.95	46.3
0	9	314.3	21.2	992.8	45.906
0	9.1	313.6	21.7	992.1	45.512
0	9.2	312.7	21.1	993	45.118
0	9.3	311.7	20.9	993.1	44.724
0	9.5	310.7	21.2	992.7	44.33
0	9.7	309.6	20.8	993.3	43.936
0	9.9	308.5	20.3	993.9	43.542

Each table row represents a hurricane center during a three-hour interval. The table includes these variables:

- `Landfall` — Indicates if the hurricane center makes landfall.
- `Latitude` and `Longitude` — Latitude-longitude coordinates of the hurricane center.
- `MaximumWindSpeed` — Maximum wind speed, in m/s.
- `MinimumPressure` — Minimum pressure, in hPa.
- `RadiusToMaximumWinds` —Radius to maximum winds, in km.
- `TCnumber` — Hurricane ID for the simulation year.
- `TimeStep` — Time interval number.
- `Year` — Simulation year. Each value of `Year` represents a simulation for 2021. When combined, the `Year` and `TCnumber` variables identify unique hurricanes.

Many Mapping Toolbox™ functions require you to use geospatial tables as input. A *geospatial table* is a table with a `Shape` variable that contains point, line, or polygon shapes. To convert the table to a geospatial table:

- Create point shapes from the latitude-longitude coordinates. Wrap the longitudes to the range [–180, 180].
- Add a `Shape` table variable that stores the point shapes.
- Remove the `Latitude` and `Longitude` table variables.

```
Shape = geopointshape(hcanes.Latitude,wrapTo180(hcanes.Longitude));
hcanes = addvars(hcanes,Shape,Before=1);
hcanes = removevars(hcanes,["Latitude","Longitude"]);
```

Prepare to visualize the hurricane data by creating a subtable of hurricane centers for an area of interest (AOI) surrounding Florida. To simplify the visualization, include only the hurricane centers that make landfall.

- Get an AOI for Florida by using the `geocode` function. The `geocode` function represents the AOI using a polygon shape in geographic coordinates.
- Ensure that the AOI contains relevant analysis data by adding a buffer of 100 km.
- Find the table rows for the hurricanes that make landfall in Florida. Create a subtable from the rows.

```
florida = geocode("Florida");
aoi = florida.Shape;

aoi = buffer(aoi,km2deg(100));

isfloridalandhcane = isinterior(aoi,Shape) & hcanes.Landfall;
floridalandhcanes = hcanes(isfloridalandhcane,:);
```

Analyze the variation in hurricane size within the AOI by displaying the minimum, maximum, and average values for `RadiusToMaximumWinds`.

```
[MinRadius,MaxRadius] = bounds(floridalandhcanes.RadiusToMaximumWinds);
MeanRadius = mean(floridalandhcanes.RadiusToMaximumWinds);
hcanestats = table(MinRadius,MaxRadius,MeanRadius);
disp(hcanestats)
```

```
    MinRadius    MaxRadius    MeanRadius
    _________    _________    __________

      9.26         203.72        52.582
```

Visualize Hurricane Data

Visualize the data by displaying the hurricane tracks on maps. Create an initial map using all the hurricane data. Then, create a more focused map by displaying the data for one sample hurricane.

Visualize All Hurricane Tracks

Create a map that uses a topographic basemap.

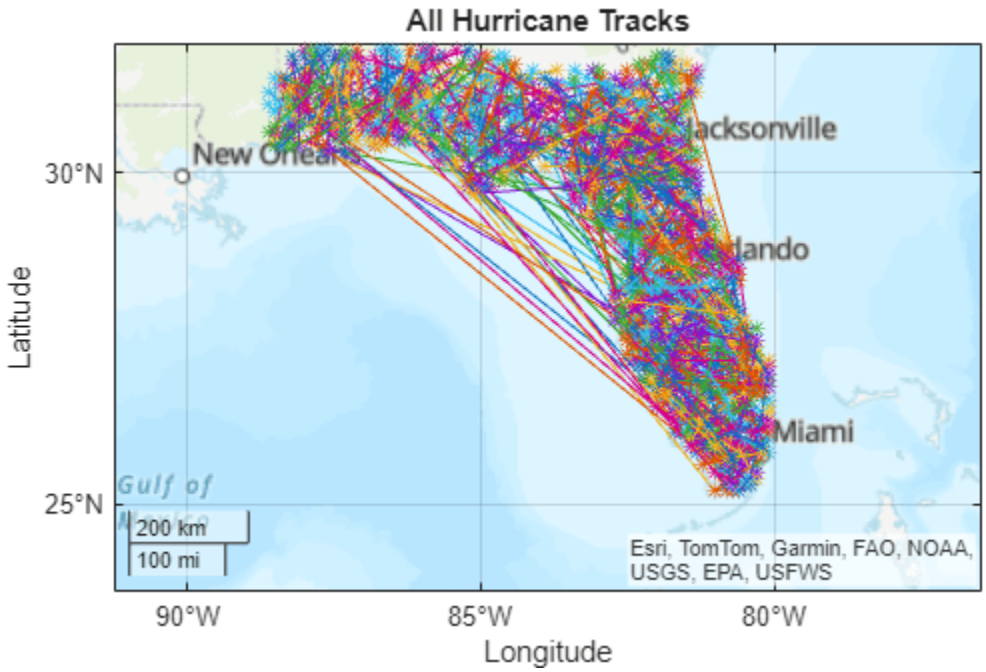
```
figure
geobasemap topographic
hold on
```

For a given hurricane track, the table rows have matching `Year` and `TCnumber` values. As a result, you can display a line for each track by iterating through the unique values of `Year` and `TCnumber`. Create the lines by using the `geoplot` function and the latitude-longitude coordinates of the hurricane centers. Mark each hurricane center with an asterisk.

```
% Iterate through unique Year values
years = unique(floridalandhcanes.Year);
for year = years'
    yearhcanes = floridalandhcanes(floridalandhcanes.Year == year,:);
    % Iterate through unique TCnumber values
    TCnums = unique(yearhcanes.TCnumber);
    for TCnum = TCnums'
        hcane = yearhcanes(yearhcanes.TCnumber == TCnum,:);
        % Plot hurricane track using a line
        geoplot(hcane.Shape.Latitude,hcane.Shape.Longitude,"-*")
    end
end
```

Set the latitude and longitude limits of the map using the bounds of the AOI. Add a title.

```
[latlim,lonlim] = bounds(aoi);
geolimits(latlim,lonlim)
title("All Hurricane Tracks")
```



Visualize Sample Hurricane

Create a subtable that contains the hurricane centers for one sample hurricane, in this case the 10th hurricane to occur during the 32nd simulation year.

```
sampleYear = 32;
sampleTCnum = 10;
sampleind = floridalandhcanes.TCnumber==sampleTCnum & floridalandhcanes.Year==sampleYear;
samplehcane = floridalandhcanes(sampleind,:);
```

Display the subtable. Note that the radius to maximum winds is approximately 55.56 km, which is close to the average radius of 52.582 km.

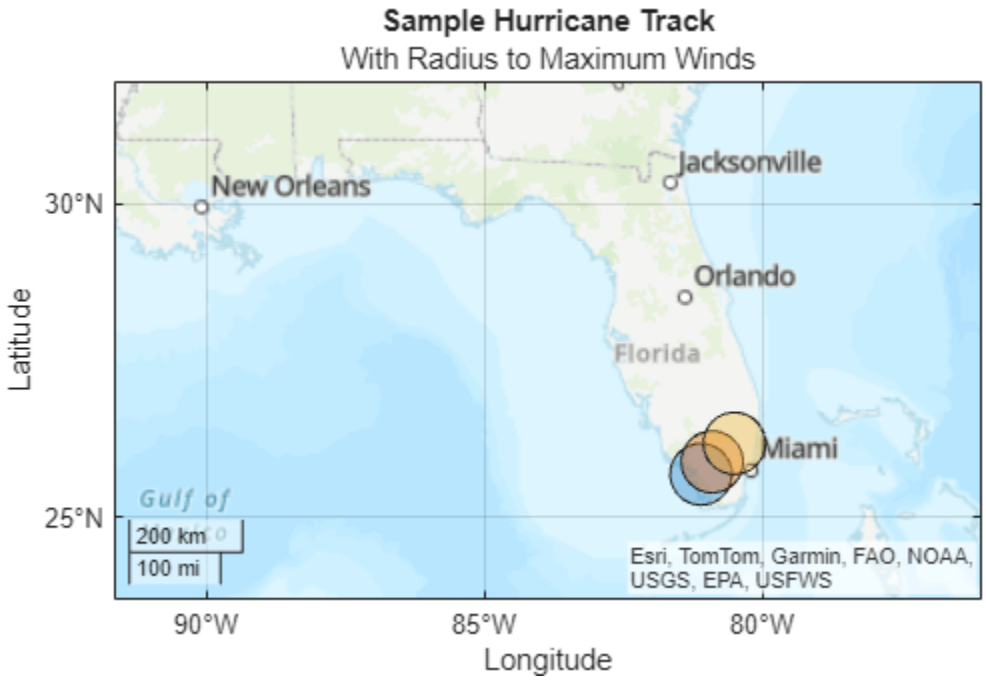
```
samplehcane
```

```
samplehcane=3×8 table
            Shape              Landfall    MaximumWindSpeed    MinimumPressure    RadiusToMaximumW
    ___________________________    ________    ________________    _______________    ________________

    (25.7000°N, 81.1000°W)         1              19.9                995                55.56
    (25.9000°N, 80.9000°W)         1              19.3                996                55.56
    (26.2000°N, 80.5000°W)         1              18.6              997.1                55.56
```

Create a new map. Then, for each time interval, create and display a circular AOI that represents the radius to maximum winds. The map shows that the sample hurricane makes landfall during three time steps.

```
figure
geobasemap topographic
hold on

for rowind = 1:height(samplehcane)
```

```
    hcanestep = samplehcane(rowind,:);
    stepcircle = aoicircle(hcanestep.Shape,km2deg(hcanestep.RadiusToMaximumWinds));
    geoplot(stepcircle)
end

geolimits(latlim,lonlim)
title("Sample Hurricane Track")
subtitle("With Radius to Maximum Winds")
```



Specify and Visualize Property Assets

Specify the exposure data using the addresses of several property assets, the geographic locations of the property assets, and the median values of surrounding property assets.

Create a list of addresses for several properties in Florida. The list includes three properties in each of these cities: Tallahassee, Tampa, Jacksonville, Gainesville, Miami, and Orlando.

```
addresses = ["911 Easterwood Dr, Tallahassee, FL" ...
    "3226 Flastacowo Rd, Tallahassee, FL" ...
    "3840 N Monroe St, Tallahassee, FL" ...
    "5300 Interbay Blvd, Tampa, FL" ...
    "3804 E 26th Ave, Tampa, FL" ...
    "11001 N 15th St, Tampa, FL" ...
    "370 Zoo Pkwy, Jacksonville, FL" ...
    "117 W Duval St, Jacksonville, FL" ...
    "3625 University Blvd S, Jacksonville, FL" ...
    "224 SE 24th St, Gainesville, FL" ...
    "803 NW 192nd Ave, Gainesville, FL" ...
    "309 Village Dr, Gainesville, FL" ...
    "11805 SW 26th St, Miami, FL" ...
    "3500 Pan American Dr, Miami, FL" ...
    "11380 NW 27th Ave, Miami, FL" ...
    "5100 L B McLeod Rd, Orlando, FL" ...
```

```
    "595 N Primrose Dr, Orlando, FL" ...
    "6501 Magic Way, Orlando, FL"];
```

Create a table that stores information about the property assets. Then, create a table row for each address by using the `geocode` function. The function represents the locations of the addresses using point shapes in geographic coordinates.

```
assets = table;
for address = addresses
    newRow = geocode(address,"address",FilterResults="first");
    assets = [assets; newRow]; %#ok<AGROW>
end
```

Prepare to calculate property damages and losses by adding two variables to the table:

- `AssetValues` estimates the median house price for each city.
- `DistanceToSea` estimates the distance, in km, from each city to the sea.

Note that, while median house prices and approximate distances are appropriate for illustrative purposes, you can create more accurate risk models by using more specific data.

```
AssetValues = zeros(numel(addresses),1);
AssetValues(1:3) = 554500;   % Tallahassee
AssetValues(4:6) = 460000;   % Tampa
AssetValues(7:9) = 301000;   % Jacksonville
AssetValues(10:12) = 315000; % Gainesville
AssetValues(13:15) = 699000; % Miami
AssetValues(16:18) = 399000; % Orlando

DistanceToSea = zeros(numel(addresses),1);
DistanceToSea(1:3) = 35;   % Tallahassee
DistanceToSea(4:6) = 5;    % Tampa
DistanceToSea(7:9) = 20;   % Jacksonville
DistanceToSea(10:12) = 80; % Gainesville
DistanceToSea(13:15) = 5;  % Miami
DistanceToSea(16:18) = 55; % Orlando

assets = addvars(assets,AssetValues,DistanceToSea);
```

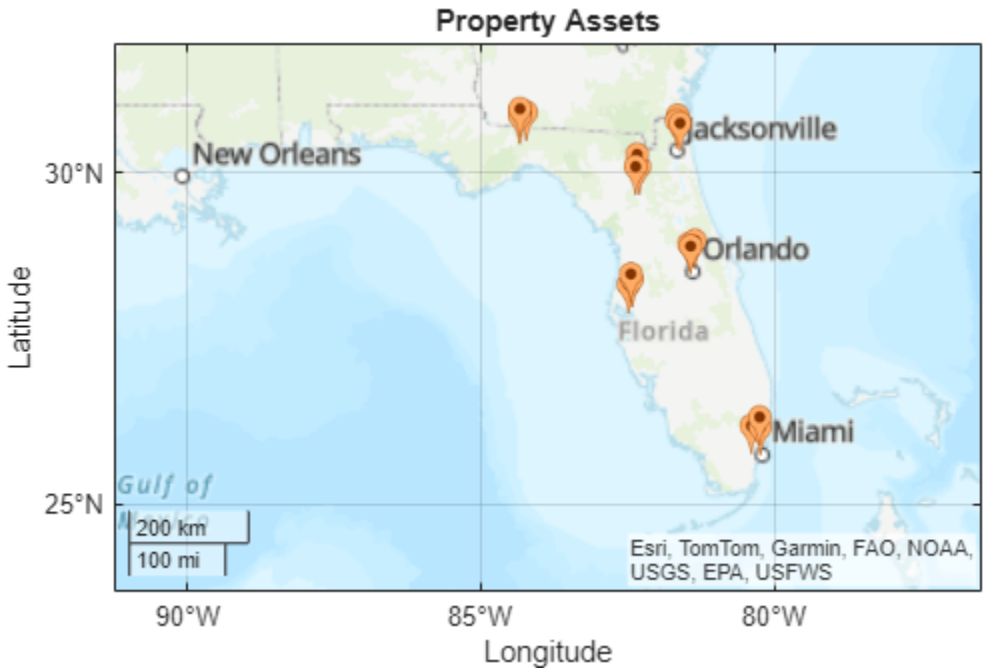
View the first two rows of the assets table.

```
head(assets,2)
```

Shape	Name	AdminLevel
(30.4515°N, 84.2227°W)	"911 Easterwood Dr, Tallahassee, FL, 32311, USA"	"address"
(30.4022°N, 84.3360°W)	"3226 Flastacowo Rd, Tallahassee, FL, 32310, USA"	"address"

Create a new map. Then, display the property assets using icons.

```
figure
geobasemap topographic
geoiconchart(assets.Shape.Latitude,assets.Shape.Longitude)
geolimits(latlim,lonlim)
title("Property Assets")
```



Define and Visualize Damage Function

Define a function that estimates property damage using the wind speeds of the hurricanes.

Define Damage Function

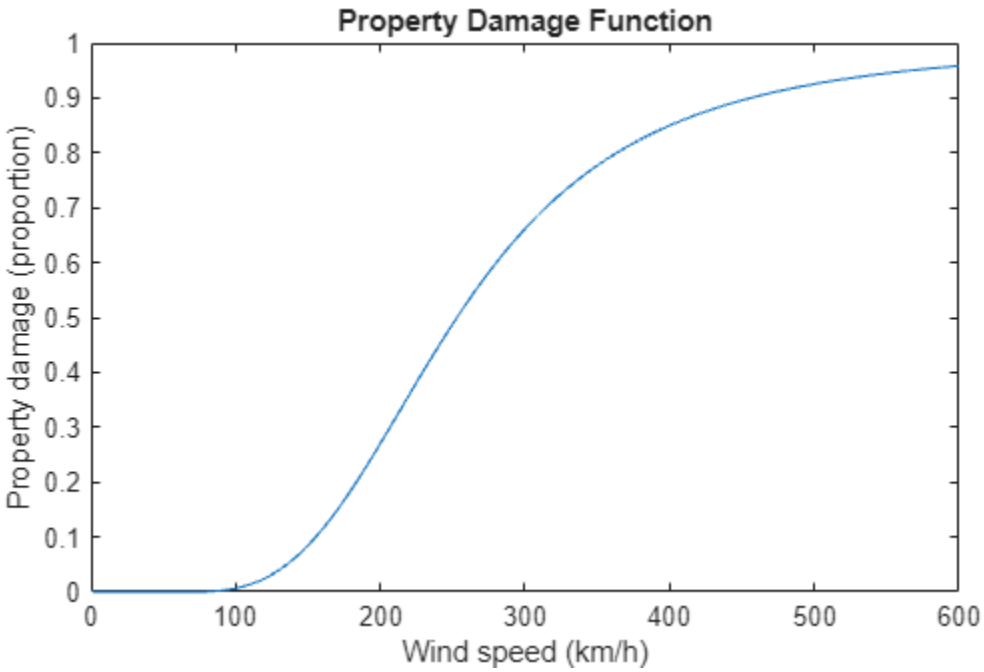
The `propertyDamage` on page 11-105 helper function uses an algorithm from [4 on page 11-107] to estimate property damage based on wind velocity and data about the property. The output of the function is the percentage of the property value that is destroyed. The inputs of the function are the wind velocity, the velocity at which the property begins to take damage, and the velocity at which half the property value is destroyed. The velocity units are km/h.

For wind velocities between 1 and 600 km/h, estimate the damage to a property asset that starts losing value at 65 km/h and loses half its value at 253 km/h. The values of `vthresh` and `vhalf` are calibration values from [3 on page 11-107].

```
v = 1:600;
vthresh = 65;
vhalf = 253;
damage = propertyDamage(v,vthresh,vhalf);
```

Plot the property damage as a function of wind speed.

```
figure
plot(v,damage)
xlabel("Wind speed (km/h)")
ylabel("Property damage (proportion)")
title("Property Damage Function")
```



Calculate Maximum Wind Speed Experienced by Assets

The `propertyDamage` function assumes that the wind velocity corresponds to the location of the property asset. The `windSpeedCalculator` helper object and `windSpeed` helper function on page 11-105 use algorithms from [6 on page 11-107] and [5 on page 11-107] to calculate the maximum wind speed experienced by a property over the course of a given hurricane.

- A `windSpeedCalculator` object stores the coordinates of the hurricane centers, the maximum wind speed values, the radius to maximum winds value, and the minimum pressure values. The hurricane data provides this information.
- The `windSpeed` function uses a `windSpeedCalculator` object, the coordinates of the property, and the distance from the property to the sea. The asset data provides this information.

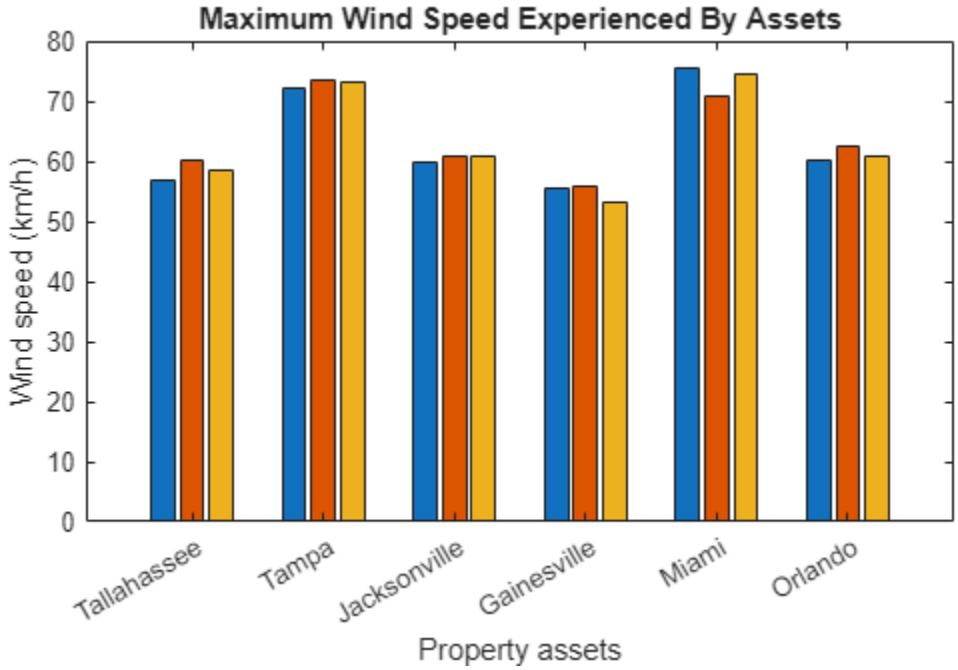
Prepare to calculate the maximum wind speeds experienced by the property assets by creating a new subtable of hurricane centers for the AOI. For this subtable, include the hurricane centers that do not make landfall.

```
isfloridahcane = isinterior(aoi,Shape);
floridahcanes = hcanes(isfloridahcane,:);
```

Using the same sample hurricane as before, calculate the maximum wind speeds experienced by the property assets.

- Using the new subtable, extract the table rows for the sample hurricane.
- Create a `windSpeedCalculator` object from the data associated with the sample hurricane.
- Use the `windSpeed` function to calculate the maximum wind speed experienced by each property asset.
- Display the maximum wind speeds using a bar graph. Each bar represents a property asset. Group the bars according to city.

```
% Get table rows for sample hurricane
sampleind = floridahcanes.TCnumber==sampleTCnum & floridahcanes.Year==sampleYear;
samplehcane = floridahcanes(sampleind,:);

% Create wind speed calculator
wsc = windSpeedCalculator(samplehcane.Shape.Latitude,samplehcane.Shape.Longitude, ...
    samplehcane.MaximumWindSpeed,samplehcane.RadiusToMaximumWinds,samplehcane.MinimumPressure);

% Calculate maximum wind speed for each property
v = windSpeed(wsc,assets.Shape.Latitude',assets.Shape.Longitude',assets.DistanceToSea');

% Create bar graph
figure
x = ["Tallahassee","Tampa","Jacksonville","Gainesville","Miami","Orlando"];
y = [v(1:3); v(4:6); v(7:9); v(10:12); v(13:15); v(16:18)];
bar(x,y)
title("Maximum Wind Speed Experienced By Assets")
xlabel("Property assets")
ylabel("Wind speed (km/h)")
```



Calculate Property Losses

Calculate the property damage for each simulated year and property asset by using the `windSpeedCalculator` object, the `windSpeed` function, and the `propertyDamage` function. This strategy assumes that the highest wind velocity experienced by a property over the course of a year determines the property damage for the year. Note that, in practice, damage functions typically include additional information about the hazard and the property.

Initialize a table that stores the property damage for each simulated year in terms of percent loss and dollar loss. Preallocate the table elements using zero values.

```
years = unique(floridahcanes.Year);
numyears = numel(years);
```

```
numassets = height(assets);
floridahcanelosses = table(zeros(numyears,1),zeros(numyears,numassets),zeros(numyears,numassets),
    VariableNames=["Year","PercentLosses","DollarLosses"]);
```

Calculate the property damage for each simulated year and hurricane. Within two loops that iterate through each simulated year and hurricane ID, perform these tasks:

- Extract the table rows that correspond to the simulated year and the hurricane ID.
- If more than one data point for the hurricane ID exists, calculate the maximum wind speed experienced by each property asset.
- Calculate the property damage for each property asset.
- Add a new row to the table that includes the year, the percent losses for each property asset, and the dollar losses for each property asset. Calculate the dollar losses using the property values from the assets table.

```
for yearInd = 1:numyears
    year = years(yearInd);
    yearhcanes = floridahcanes(floridahcanes.Year == year,:);

    TCnums = unique(yearhcanes.TCnumber);
    v = zeros(1,numassets);
    for TCnum = TCnums'
        % Get table rows that correspond to year and ID
        hcane = yearhcanes(yearhcanes.TCnumber == TCnum,:);

        if height(hcane) > 1
            % Calculate maximum wind speed experienced by each property
            wsc = windSpeedCalculator(hcane.Shape.Latitude,hcane.Shape.Longitude, ...
                hcane.MaximumWindSpeed,hcane.RadiusToMaximumWinds,hcane.MinimumPressure);
            v2 = windSpeed(wsc,assets.Shape.Latitude',assets.Shape.Longitude',assets.DistanceToSe
            v = max(v,v2);
        end
    end

    % Calculate property damage
    damage = propertyDamage(v,vthresh,vhalf);

    % Store results in table
    floridahcanelosses.Year(yearInd) = year;
    floridahcanelosses.PercentLosses(yearInd,:) = damage*100;
    floridahcanelosses.DollarLosses(yearInd,:) = damage.*assets.AssetValues';
end
```

The table of property damages only includes rows for years where hurricanes pass through the AOI. Prepare to calculate average outcomes by creating a new table that contains a row for each simulated year.

- Initialize a new table that includes a row for each simulated year. Preallocate the table elements using zero values.
- Iterate through each simulated year. If the table of property damages includes a row for the year, then copy the row to the new table. Otherwise, keep the zero values.

```
% Initialize new table
allyears = unique(hcanes.Year);
numallyears = numel(allyears);
hcanelosses = table(zeros(numallyears,1),zeros(numallyears,numassets),zeros(numallyears,numassets
```

```
    VariableNames=["Year","PercentLosses","DollarLosses"]);

% Iterate through each year
for yearInd = 1:numel(allyears)
    year = allyears(yearInd);
    if ismember(year,years)
        hcanelosses(yearInd,:) = floridahcanelosses(floridahcanelosses.Year == year,:);
    else
        hcanelosses.Year(yearInd) = year;
    end
end
```

Analyze Property Losses

The table `hcanelosses` stores the property damage for each simulated year in terms of percent loss and dollar loss. Analyze the results by viewing the losses for a selected property and for a portfolio of all the properties.

Query Losses for Selected Property

Select a property from the assets table by specifying a row number. Display the corresponding row of the table.

```
propertyNum = 2 ;
disp(assets(propertyNum,:))
```

Shape	Name	AdminLevel
(30.4022°N, 84.3360°W)	"3226 Flastacowo Rd, Tallahassee, FL, 32310, USA"	"address"

For each simulated year, query the percent and dollar losses for the property.

```
percentLosses = hcanelosses.PercentLosses(:,propertyNum);
dollarLosses = hcanelosses.DollarLosses(:,propertyNum);
```

Calculate and display the largest loss experienced by the property.

```
maxPercentLoss = max(percentLosses);
maxDollarLoss = max(dollarLosses);
fprintf("Largest Percent Loss: " + maxPercentLoss + "\n" + "Largest Dollar Loss: " + maxDollarLos
```

```
Largest Percent Loss: 8.2338
Largest Dollar Loss: 45656.2922
```

Calculate and display the expected loss per year by calculating the average of the losses.

```
meanPercentLoss = mean(percentLosses);
meanDollarLoss = mean(dollarLosses);
fprintf("Expected Percent Loss: " + meanPercentLoss + "\n" + "Expected Dollar Loss: " + meanDolla
```

```
Expected Percent Loss: 0.059618
Expected Dollar Loss: 330.5824
```

For risk management purposes, a more realistic metric is percentile loss. Calculate and display the dollar loss for the property at the 95th percentile.

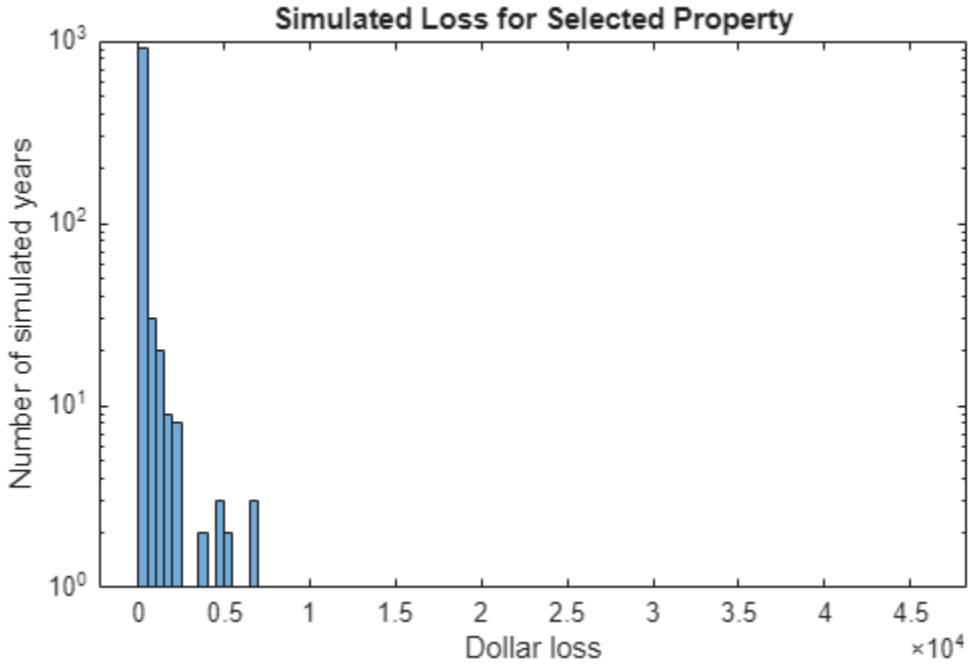
```
percentile = 0.95;
percentileDollarLoss = quantile(dollarLosses,percentile);
fprintf("Dollar Loss at 95th Percentile: " + percentileDollarLoss)
```

```
Dollar Loss at 95th Percentile: 1266.2186
```

View Loss Distribution for Selected Property

View the distribution of the dollar losses by creating a histogram. Improve the visibility of the tail by changing the *y*-axis scale to logarithmic.

```
figure
histogram(dollarLosses)
title("Simulated Loss for Selected Property")
ylabel("Number of simulated years")
xlabel("Dollar loss")
yscale log
```



Query Losses for All Properties

Analyze the NatCat risk for a portfolio of all the property assets. For each simulated year, sum the losses for all the properties. Then, calculate the dollar losses for the entire portfolio at the 95th percentile.

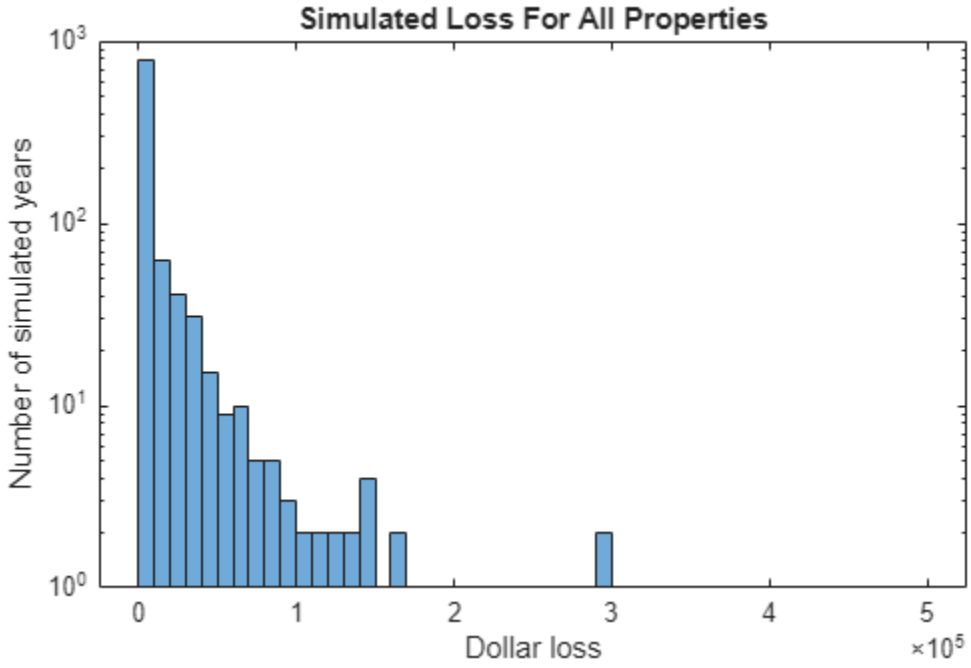
```
yearlyTotalDollarLosses = sum(hcanelosses.DollarLosses,2);
yearlyTotalDollarPercentileLosses = quantile(yearlyTotalDollarLosses,percentile);
fprintf("Total Dollar Loss at 95th Percentile: " + yearlyTotalDollarPercentileLosses)
```

```
Total Dollar Loss at 95th Percentile: 60899.9064
```

View Loss Distribution for All Properties

View the distribution of the dollar losses for the portfolio by creating a new histogram.

```
figure
histogram(yearlyTotalDollarLosses)
title("Simulated Loss For All Properties")
ylabel("Number of simulated years")
xlabel("Dollar loss")
yscale log
```



Display Total Property Losses on Map

Add a new variable to the assets table that stores the total dollar loss for each property at the 95th percentile.

```
TotalPercentileLosses = quantile(hcanelosses.DollarLosses,percentile)';
assets = addvars(assets,TotalPercentileLosses);
```

Display each property on a new map. Indicate the total dollar loss for each property asset by applying a color to the corresponding marker. Add a labeled color bar, change the geographic limits, and add a title.

```
figure
geobasemap topographic
geoplot(assets,ColorVariable="TotalPercentileLosses",MarkerSize=20)

c = colorbar;
c.Label.String = "Dollars";

geolimits(latlim,lonlim);
title("Total Property Losses")
```

Helper Functions

`propertyDamage` Function

The `propertyDamage` function estimates property damage based on the specified wind velocity and property data. From [4 on page 11-107], the percent of the property value that is destroyed D is:

$$D = \frac{W^3}{1 + W^3}, \text{ where:}$$

$$W = \frac{\max([V - V_{\text{thresh}}], 0)}{V_{\text{half}} - V_{\text{thresh}}},$$

and V is the wind velocity, V_{thresh} is the velocity at which the property begins to take damage, and V_{half} is the velocity at which half of the property value is destroyed.

```
function D = propertyDamage(v,vthresh,vhalf)
    k = size(v,2);
    v = max(v - vthresh,zeros(1,k))./(vhalf - vthresh);
    D = (v.^3)./(1 + v.^3);
end
```

`windSpeedCalculator` Object and `windSpeed` Function

The `windSpeedCalculator` object and the `windSpeed` function calculate the maximum wind speed experienced by a property asset over the course of a hurricane. The object and function are attached to the example as a supporting file.

- For a given hurricane, a `windSpeedCalculator` object stores the coordinates of the hurricane centers, the maximum wind speed values, the radius to maximum winds value, and the minimum pressure values. The example specifies these inputs using the hurricane data from [2 on page 11-107].

- For a given property asset, the inputs to `windSpeed` are a `windSpeedCalculator` object, the coordinates of the property, and the distance from the property to the sea. The example specifies these inputs using the asset data stored in `assets`.

From [6 on page 11-107], the maximum wind velocity V_i experienced by a property asset over the course of a given hurricane is:

$$V_i = \mathrm{GF}\left[V_m - S(1 - \sin(T_i))\frac{V_h}{2}\right] \times \left[\left(\frac{R_m}{R_i}\right)^B \exp\left(1 - \left[\frac{R_m}{R_i}\right]^B\right)\right]^{\frac{1}{2}},$$

where:

- $G = 1.5$ is the gust factor, from [6].
- $F = 0.7(1 - R)$ is the friction parameter, from [6]. R is a reduction factor that scales linearly as a function of distance inland. The reduction factor starts at 0.14 and scales to 0.28 at 50 km from the coast.
- $S = 1$ is the storm asymmetry factor, from [6].
- V_m is the maximum wind velocity in km/h, from the hurricane data.
- T_i is the clockwise angle made by the forward direction of the hurricane and the line made by the property asset and the hurricane center. The `windSpeed` function calculates the forward direction of the hurricane from the hurricane data.
- V_h is the forward velocity of the hurricane in km/h. The `windSpeed` function calculates V_h from the hurricane data.
- R_m is the radius to maximum winds, from the hurricane data.
- R_i is the radius to the property asset in km. The `windSpeed` function calculates R_i from the hurricane data and the asset data.
- B is the wind profile parameter, defined by [5].

From [5 on page 11-107], the wind profile parameter B is $B = 1.6 \times B_s$, where:

$$B_s = -4.4 \times 10^{-5}\Delta p^2 + 0.01\Delta p + 0.03\frac{\partial p_c}{\partial t} - 0.014\phi + 0.15V_t^x + 1,$$

and:

- $x = 0.6 \times \left(1 - \frac{\Delta p}{215}\right)$.
- $\Delta p = p_n - p_c$.
- $p_n = 1016$ is the normal atmospheric pressure in hPa, from [5].
- p_c is the atmospheric pressure in the center of the hurricane in hPa, from the hurricane data.
- $\frac{\partial p_c}{\partial t}$ is the change in central pressure over time in hPa/h. The `windSpeed` function calculates this value from the hurricane data.
- ϕ is the absolute value of the latitude of the property, from the hurricane data.
- V_t is the forward velocity of the hurricane in m/s. The `windSpeed` function calculates V_t from the hurricane data.

References

[1] Bloemendaal, Nadia, Ivan D. Haigh, Hans De Moel, Sanne Muis, Reindert J. Haarsma, and Jeroen C. J. H. Aerts. "Generation of a Global Synthetic Tropical Cyclone Hazard Dataset Using STORM." *Scientific Data* 7, no. 1 (February 6, 2020): 40. https://doi.org/10.1038/s41597-020-0381-2.

[2] Bloemendaal, Nadia, I.D. (Ivan) Haigh, H. (Hans) de Moel, S Muis, R.J. (Reindert) Haarsma, and J.C.J.H. (Jeroen) Aerts. "STORM IBTrACS Present Climate Synthetic Tropical Cyclone Tracks." Media types: application/vnd.google-earth.kml+xml, application/zip, text/plain. 4TU.Centre for Research Data, September 30, 2020. https://doi.org/10.4121/UUID:82C1DC0D-5485-43D8-901A-CE7F26CDA35D.

[3] Bressan, Giacomo, Anja Duranovic, Irene Monasterolo, and Stefano Battiston. “Asset-Level Climate Physical Risk Assessment and Cascading Financial Losses.” *SSRN Electronic Journal*, 2022. https://doi.org/10.2139/ssrn.4062275.

[4] Emanuel, Kerry. "Global Warming Effects on U.S. Hurricane Damage." *Weather, Climate, and Society* 3, no. 4 (October 1, 2011): 261–68. https://doi.org/10.1175/WCAS-D-11-00007.1.

[5] Holland, Greg. "A Revised Hurricane Pressure–Wind Model." *Monthly Weather Review* 136, no. 9 (September 1, 2008): 3432–45. https://doi.org/10.1175/2008MWR2395.1.

[6] Ishizawa, Oscar A., Juan José Miranda, and Eric Strobl. "The Impact of Hurricane Strikes on Short-Term Local Economic Activity: Evidence from Nightlight Images in the Dominican Republic." *International Journal of Disaster Risk Science* 10, no. 3 (September 2019): 362–70. https://doi.org/10.1007/s13753-019-00226-0.

[7] Foote, Matthew, John Hillier, Kirsten Mitchell-Wallace, and Matthew Jones. *Natural Catastrophe Risk Management and Modelling: A Practitioner's Guide*. 1st ed. Wiley, 2017. https://doi.org/10.1002/9781118906057.

See Also

Functions

`geocode` | `geoplot` | `geoiconchart`

Related Examples

- “Create Geospatial Tables” on page 6-2
- “Define Areas of Interest” on page 6-82
- “Assess Physical and Transition Risk for Mortgages” (Risk Management Toolbox)