ADAS Algorithm Design and Prototyping
Forward Collision Warning Example

Seo-Wook Park, Principal Application Engineer
Mark Corless, Principal Application Engineer
How do I know my ADAS algorithm is robust enough?

- OEM specific test scenarios
- Fail Operation test scenarios
- Scenarios identified from real world test drive data

- Worst-case scenarios
Example workflow for ADAS algorithm development

1. Synthetic data
2. Logged vehicle data
3. ADAS algorithm
   - Expected behavior?
     - yes: Generate code
     - no: Refine algorithm
6. Drive and collect more vehicle data
5. Create new traffic scenario or refine sensor model
4. Integrate with embedded environment

C Code
The MATLAB environment helps ADAS engineers …

- Gain insight by *replaying and visualizing* logged vehicle data
- Reduce time on the road by *synthesizing* data to test algorithms
- Speed up prototyping of algorithms by *generating code*
Gain insight by *replaying and visualizing* logged vehicle data

1. **Synthetic data**
   - Logged vehicle data
   - ADAS algorithm
   - Expected behavior?
     - yes
     - no
     - Refine algorithm
   - Drive and collect more vehicle data
   - Create new traffic scenario or refine sensor model

2. **Generate code**
   - C Code

3. **Integrate with embedded environment**
Test vehicle equipped with sensors

Velodyne LiDAR HDL-32E
- Horizontal FoV: 360°
- Vertical FoV: +10°..-30°
- Range: 80..100m
- 100 Mbps Ethernet

Mobileye 560
- FoV: 38°x150m
- CAN interface

Point Grey Blackfly
- Stand “ice cube” vision camera
- 800x600, 27FPS
- GigE interface

XSENS MTI-G-700
- Stable and sensitive
- MEMS-based AHRS
- USB interface

Delphi ESR
- 76GHz electronically scanning radar
- Dual FoVs, 90°x60m, 20°x174m
- CAN interface
Example test scenarios in public road

- 01_city_c2s_fcw
- 02_city_stopngo
- 03_local_streetParking
- 04_highway_cornering
- 05_highway_lanechange
- 06_highway_cutin
Sensor fusion algorithm for FCW

Data Pre-processing

Path Estimation

Sensor Fusion & Tracking

Threat Assessment

Radar Object

Vision Object

Vision LD

Vehicle CAN

MIO: Most-Important Object
Sensor fusion algorithm for FCW

- **Data Pre-processing**
  - Calculate Ground Speed
  - Object classification
  - Filtering
  - Offset Compensation

- **Path Estimation**
  - Zoning

- **Sensor Fusion & Tracking**
  - Sensor Fusion
  - Kalman Filter

- **Threat Assessment**
  - Maneuver Analysis
  - Risk Assessment
  - Find MIO

**MIO**: Most-Important Object
Sensor fusion algorithm for FCW

Data Pre-processing:
- Calculate Ground Speed
- Object classification

Path Estimation:
- Zoning

Sensor Fusion & Tracking:
- Sensor Fusion
- Kalman Filter

Threat Assessment:
- Maneuver Analysis
- Risk Assessment
- Find MIO

MIO: Most-Important Object
Sensor fusion

Birds-Eye View object notations
- Radar object (stationary)
- Radar object (moving)
- Vision object
- Fused object (safe zone)
- Fused object (warn zone)
- Fused object (FCW zone)
- Fused object (most important object)
Sensor Fusion Made Easy by MATLAB CVST

Computer Vision System Toolbox™

![Diagram](image)

Pairs of visions and associated radars

```matlab
[assignments, unassignedVisions, unassignedRadars] = ...
assignDetectionsToTracks(costMatrix, param.costOfNonAssignment);
```
Object Tracking by Kalman Filter

Data Pre-processing
- Calculate Ground Speed
- Object classification

Path Estimation
- Zoning

Sensor Fusion & Tracking
- Sensor Fusion
- Kalman Filter

Threat Assessment
- Maneuver Analysis
- Risk Assessment
- Find MIO

MIO: Most-Important Object

Radar Object
Vision Object
Vision LD
Vehicle CAN

Kalman Filter
Path Estimation

Find MIO
FCW

Filtering
Offset Compensation
What is the Kalman Filter?

- It is an iterative mathematical process that uses a set of equations and consecutive data inputs to quickly estimate the true value, position, velocity, etc. of the object being measured, when the measured values contain random noise.
Kalman Filter

Initial state & covariance

Output of updated state

Previous state & covariance

Time Update ("Predict")

1. Predict state based on physical model and previous state
   \[
   \hat{x}_k^- = A \hat{x}_{k-1} + B u_k + w_k
   \]
   \[
   P_k^- = AP_{k-1}A^T + Q
   \]

Measurement Update ("Correct")

1. Compute Kalman gain
   \[
   K_k = P_k^- H^T (HP_k^- H^T + R)^{-1}
   \]
   \[
   \hat{x}_k = \hat{x}_k^- + K_k (z_k - H \hat{x}_k^-)
   \]
   \[
   P_k = (I - K_k H) P_k^-
   \]

- Time Update ("Predict")
- Measurement Update ("Correct")

\[ u \text{ : Control variable matrix} \]
\[ w \text{ : Process (state) noise} \]
\[ P_k^- = E[e_k e_k^T] \text{ : Process (state) covariance matrix} \]
\[ e_k^- = x_k - \hat{x}_k^- \text{ : Covariance matrix (estimation error)} \]
\[ Q = E[ww^T] \text{ : Process noise covariance matrix} \]
\[ A \text{ : State matrix relates the state at the previous, } k-1 \text{ to the state at the current, } k \]
\[ \hat{x}_k \text{ : Output of updated state} \]
\[ P_k \text{ : Current becomes previous} \]
\[ z_k = H x_k + v_k \text{ : Measurement} \]
\[ v \text{ : Measurement noise} \]
\[ H \text{ : Output matrix relates the state to the measurement} \]
\[ \hat{x}_k \text{ : State vector} \]
\[ P_k \text{ : Error covariance matrix} \]
\[ \hat{x}_k^- \text{ : Prev state estimate} \]
\[ e_k^- \text{ : Estimation error} \]
\[ z_k \text{ : Measurement} \]
\[ v_k \text{ : Measurement noise} \]
\[ \hat{x}_k \text{ : Updated state estimate} \]
Kalman Filter Made Easy by MATLAB CVST

Initial state & covariance
\[ \hat{x}_0 \quad P_0 \]

Previous state & covariance
\[ \hat{x}_{k-1} \quad P_{k-1} \]

\( k \rightarrow k - 1 \)
Current becomes previous

Time Update ("Predict")

\[ [z_{\text{pred}}, x_{\text{pred}}, P_{\text{pred}}] = \text{predict}(\text{obj}) \]

\( z_{\text{pred}} \): prediction of measurement
\( x_{\text{pred}} \): prediction of state
\( P_{\text{pred}} \): state estimation error covariance at the next time step

Measurement Update ("Correct")

\[ [z_{\text{corr}}, x_{\text{corr}}, P_{\text{corr}}] = \text{correct}(\text{obj}, z) \]

\( z_{\text{corr}} \): correction of measurement
\( x_{\text{corr}} \): correction of state
\( P_{\text{corr}} \): state estimation error covariance

\[ \text{kalmanFilterSysObj} = \text{vision.KalmanFilter}(A,H,'ProcessNoise',Q,'MeasurementNoise',R) \]
Sensor fusion algorithm for FCW

Data Pre-processing
- Calculate Ground Speed
- Object classification

Path Estimation
- Zoning

Sensor Fusion & Tracking
- Sensor Fusion
- Kalman Filter
- Path Estimation

Threat Assessment

- Radar Object
- Vision Object
- Vision LD
- Vehicle CAN

Sensor fusion algorithm for FCW
Reduce time on the road by *synthesizing* data to test algorithms

- Synthetic data
  - Logged vehicle data
  - ADAS algorithm
  - Expected behavior?
    - yes
    - no
    - Refine algorithm
    - Drive and collect more vehicle data
    - Create new traffic scenario or refine sensor model
  - Generate code
    - C Code
    - Integrate with embedded environment
Example synthetic test scenarios – EuroNCAP

- ENCAP – CCRs
  - Stationary
  - 50kph

- ENCAP – CCRm
  - 20kph
  - 50kph

- ENCAP – CCRb
  - 50kph @ -2m/s²
  - 12m

- ENCAP – CCRb
  - 50kph @ -6m/s²
  - 40m

Ego car
Target car
Example synthetic test scenarios – Fail Operation

- **FO1 – cornering**: 30kph, 60kph
- **FO2 – overtaking**: 90kph, 130kph
- **FO4 – lane change**: stationary, 50kph
Example synthetic test scenarios – intersection and stop & go

- Intersection – turning w/ braking
- Intersection – near crash
- Intersection – U-turn
- Stop & go @ 30→20kph

50kph

Ego car
Target car
Common approaches to synthesizing test data

“Buy It”
- Buying “Off the shelf” solutions like PreScan and CarMaker enable you to author traffic scenarios and synthesize sensor data

“Build It”
- Building it yourself enables you to control the level of fidelity/complexity appropriate for your application
- If building for multiple users, it is important to select an approach which is scalable, maintainable, and testable
Key components defining a Traffic Scenario

Define Road
- Road type
- NumLanes
- LaneWidth

Define Vehicles
- Ego vehicle
- Target vehicles
- NumTargets

Define Trajectories
- Path type
- Waypoints
- Velocity, acceleration
- Heading angle, yawrate

Sensor Model
- Vision, Radar
- Range, FoV
- Position, Velocity error
- NumCluster
Example radar spec

<table>
<thead>
<tr>
<th>Parameter</th>
<th>ESR Long-Range Requirement</th>
<th>ESR Mid-Range Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Range</td>
<td>&lt; 1m (&gt; 10dB target)</td>
<td>&lt; 1m (&gt; 10dB target)</td>
</tr>
<tr>
<td></td>
<td>&lt; 1m (&gt; 0dB target)</td>
<td>&lt; 1m (&gt; 0dB target)</td>
</tr>
<tr>
<td>Maximum Range</td>
<td>&gt; 175m (&gt; 10dB target)</td>
<td>&gt; 60m (&gt; 10dB target)</td>
</tr>
<tr>
<td></td>
<td>&gt; 100m (&gt; 0dB target)</td>
<td>&gt; 50m (&gt; 0dB target)</td>
</tr>
<tr>
<td>Range Accuracy</td>
<td>&lt; +/-0.5m noise component with +/- 5% bias component</td>
<td>&lt; +/-0.25m noise component with +/- 5% bias component</td>
</tr>
<tr>
<td>Range Discrimination for two targets at same angle &amp; range rate</td>
<td>&lt; 2.5m</td>
<td>&lt; 1.25m</td>
</tr>
<tr>
<td>Minimum Range Rate</td>
<td>&lt; -100m/s</td>
<td></td>
</tr>
<tr>
<td>Maximum Range Rate</td>
<td>&gt; +40m/s</td>
<td></td>
</tr>
<tr>
<td>Range Rate Accuracy</td>
<td>&lt; +/- 0.12m/s</td>
<td></td>
</tr>
</tbody>
</table>

Architecture we selected to build a data synthesis tool

1. Data Model (Programmatic Interface)
   - Controls state of data
     - Specifies scenario (i.e. road, vehicles, waypoints, sensor models)
     - Synthesizes data to test algorithm
     - Enables automation

Synthesized Data
Architecture we selected to build a data synthesis tool

1. Data Model (Programmatic Interface)
   - Controls state of data
     - Specifies scenario (i.e. road, vehicles, waypoints, sensor models)
     - Synthesizes data to test algorithm
     - Enables automation

2. App (Graphical Interface)
   - Controls state of interface
     - Reacts to user input
     - Updates data model
     - Draws user interface and visualizations
     - Simplifies interaction with data

Synthesized Data
Architecture we selected to build a data synthesis tool

1. Data Model (Programmatic Interface)
   - Controls state of data
     - Specifies scenario (e.g., road, vehicles, waypoints, sensor models)
     - Synthesizes data to test algorithm
     - Enables automation

2. App (Graphical Interface)
   - Controls state of interface
     - Reacts to user input
     - Updates data model
     - Draws user interface and visualizations
     - Simplifies interaction with data

3. Unit Test
   - Automates regression testing
     - Increases likelihood of catching issues introduced during iterations
     - Facilitates collaboration

Synthesized Data
Architecture we selected to build a data synthesis tool

1. Data Model (Programmatic Interface)
   - Controls state of data
     - Specifies scenario (i.e. road, vehicles, waypoints, sensor models)
     - Synthesizes data to test algorithm
     - Enables automation

2. App (Graphical Interface)
   - Controls state of interface
     - Reacts to user input
     - Updates data model
     - Draws user interface and visualizations
     - Simplifies interaction with data

3. Unit Test
   - Automates regression testing
     - Increases likelihood of catching issues introduced during iterations
     - Facilitates collaboration

Synthesized Data
Speed up prototyping of algorithms by *generating code*

1. Synthetic data
2. ADAS algorithm
3. Expected behavior?
   - yes: Generate code
   - no:
     - Logged vehicle data
     - Refine algorithm
     - Drive and collect more vehicle data
     - Create new traffic scenario or refine sensor model
4. Integrate with embedded environment

C Code
Generate C code for your algorithm with MATLAB Coder

```matlab
function [mostImportantObject, ... 
    egoPath, ... 
    radarObjects, ... 
    visionObjects, ... 
    fusedObjects, ... 
    assessedThreats] = forwardCollisionWarning( ... 
    lane, ... 
    inertialMeasurementUnit, ... 
    visionSensor, ... 
    radarSensor, ... 
    params, ... 
    reset)
```

```c
void forwardCollisionWarning(const laneStruct *lane, const 
    inertialMeasurementUnitStruct *inertialMeasurementUnit, const 
    visionObjectsStruct *visionSensor, const radarObjectsStruct *radarSensor, 
    const fcwParamsStruct *params, boolean_T reset, mostImportantObjectStruct 
    *mostImportantObject, egoPathStruct *egoPath, radarObjectsStruct *radarObjects, 
    visionObjectsStruct *visionObjects, struct3_T *fusedObjects, struct2_T 
    *assessedThreats)
```

Software in the loop & Processor in the loop enabled by Embedded Coder
Create components in MATLAB and reuse them in Simulink

```matlab
[mostImportantObject, ... egoPath, ... radarObjects, ... visionObjects, ... fusedObjects, ... assessedThreats] = forwardCollisionWarning( ... lane, inertialMeasurementUnit, ... vision, radar, params, reset);
```
Enable coverage of MATLAB and C code with Simulink Verification and Validation

Collects coverage on MATLAB code in “Normal” mode

Collects coverage on generated C code in “Software in the loop” mode
Automate regression testing with Simulink Test

Specify tests and Interact with results in the Test Manager

Generate a report to share results
Prototype on hardware with Simulink Real-Time
as seen in today’s Test drive your ADAS algorithms presentation

Algorithm Models

- Forward Collision Warning
- Autonomous Emergency Braking

Vehicle and Environment Models
Example workflow for ADAS algorithm development

1. Synthetic data
2. Logged vehicle data
3. ADAS algorithm
4. Expected behavior?
   - yes
   - no
5. Generate code
6. C Code
7. Integrate with embedded environment
8. Refine algorithm
9. Drive and collect more vehicle data
10. Create new traffic scenario or refine sensor model
What key product features enabled this workflow?

- **Design Kalman filters** with Computer Vision System Toolbox
- **Encapsulate data** with MATLAB Classes
- **Test MATLAB code** with MATLAB Unit Test
- **Generate C code** from your algorithm with MATLAB Coder
- **Verify generated code** using SIL with Embedded Coder
- **Collect model and code coverage** with Simulink Verification and Validation
- **Automate regression testing** and reporting with Simulink Test
How can you get started?

Consulting Services can help you select and establish a baseline architecture to meet the needs of your application.

Consulting helped us select an architecture for our Traffic Scenario App.

Documentation and Training Services can help you learn how to use product features.
The MATLAB environment helps ADAS engineers …

- Gain insight by *replaying and visualizing* logged vehicle data
- Reduce time on the road by *synthesizing* data to test algorithms
- Speed up prototyping of algorithms by *generating code*