

Navigation Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2022a



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

Navigation Toolbox™ User's Guide

© COPYRIGHT 2019–2022 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

September 2019	Online only	New for Version 1.0 (R2019b)
March 2020	Online only	Rereleased for Version 1.1 (R2020a)
September 2020	Online only	Revised for Version 1.2 (R2020b)
March 2021	Online only	Revised for Version 2.0 (R2021a)
September 2021	Online only	Revised for Version 2.1 (R2021b)
March 2022	Online only	Revised for Version 2.2 (R2022a)

1

Navigation Featured Examples

Optimization Based Path Smoothing for Autonomous Vehicles	1-3
Benchmark Path Planners for Differential Drive Robots in Warehouse Map	1-14
Generate Code for Path Planning Using Hybrid A Star	1-30
Estimate Position and Orientation of a Ground Vehicle	1-35
Pose Estimation From Asynchronous Sensors	1-43
Inertial Sensor Noise Analysis Using Allan Variance	1-48
Simulate Inertial Sensor Readings from a Driving Scenario	1-59
Estimate Orientation Through Inertial Sensor Fusion	1-67
IMU and GPS Fusion for Inertial Navigation	1-77
GNSS Simulation Overview	1-85
Estimate GNSS Receiver Position with Simulated Satellite Constellations	1-93
Analyze GPS Satellite Visibility	1-99
Simulate GPS Sensor Noise	1-103
IMU Sensor Fusion with Simulink	1-105
Automatic Tuning of the insfilterAsync Filter	1-107
Custom Tuning of Fusion Filters	1-115
Magnetometer Calibration	1-125
Wheel Encoder Error Sources	1-134
Remove Bias from Angular Velocity Measurement	1-141
Detect Multipath GPS Reading Errors Using Residual Filtering in Inertial Sensor Fusion	1-145

Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter	1-151
Estimate Orientation with a Complementary Filter and IMU Data	1-155
Estimate Orientation Using AHRS Filter and IMU Data in Simulink . .	1-163
Estimate Phone Orientation Using Sensor Fusion	1-172
Binaural Audio Rendering Using Head Tracking	1-178
Estimating Orientation Using Inertial Sensor Fusion and MPU-9250 .	1-183
Wireless Data Streaming and Sensor Fusion Using BNO055	1-193
Rotations, Orientation, and Quaternions	1-199
Lowpass Filter Orientation Using Quaternion SLERP	1-214
Introduction to Simulating IMU Measurements	1-218
Logged Sensor Data Alignment for Orientation Estimation	1-230
Generate Off-Centered IMU Readings	1-237
Estimate Robot Pose with Scan Matching	1-242
Localize TurtleBot Using Monte Carlo Localization	1-248
Compose a Series of Laser Scans with Pose Changes	1-262
Minimize Search Range in Grid-based Lidar Scan Matching Using IMU	1-266
Visual-Inertial Odometry Using Synthetic Data	1-271
Landmark SLAM Using AprilTag Markers	1-280
Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs . .	1-292
Create Egocentric Occupancy Maps Using Range Sensors	1-295
Build Occupancy Map from Lidar Scans and Poses	1-300
Create Egocentric Occupancy Map from Driving Scenario Designer . .	1-301
Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph	1-307
Fuse Multiple Lidar Sensors Using Map Layers	1-310
Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans	1-319

Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans	1-326
Perform SLAM Using 3-D Lidar Point Clouds	1-333
Plan Mobile Robot Paths Using RRT	1-341
Moving Furniture in a Cluttered Room with RRT	1-347
Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map	1-352
Motion Planning with RRT for a Robot Manipulator	1-366
Dynamic Replanning on an Indoor Map	1-372
Highway Lane Change	1-378
Path Following with Obstacle Avoidance in Simulink®	1-391
Obstacle Avoidance with TurtleBot and VFH	1-395
Highway Trajectory Planning Using Frenet Reference Path	1-397
Optimal Trajectory Generation for Urban Driving	1-411
EKF-Based Landmark SLAM	1-425
Reverse-Capable Motion Planning for Tractor-Trailer Model Using plannerControlRRT	1-430
Simulate INS Block	1-445
Object Tracking and Motion Planning Using Frenet Reference Path ..	1-447
Offroad Planning with Digital Elevation Models	1-458
Factor Graph-Based Pedestrian Localization with IMU and GPS Sensors	1-478
Ground Vehicle Pose Estimation for Tightly Coupled IMU and GNSS .	1-483
Estimate Orientation Using GPS-Derived Yaw Measurements	1-488
Design Fusion Filter for Custom Sensors	1-494

Navigation Topics

2

Model IMU, GPS, and INS/GPS	2-2
Inertial Measurement Unit	2-2

Global Positioning System	2-4
Inertial Navigation System and Global Positioning System	2-6
Choose Inertial Sensor Fusion Filters	2-8
Configure Time Scope MATLAB Object	2-13
Signal Display	2-13
Multiple Signal Names and Colors	2-14
Configure Scope Settings	2-14
Use timescope Measurements and Triggers	2-15
Share or Save the Time Scope	2-31
Scale Axes	2-32
Fuse Inertial Sensor Data Using insEKF-Based Flexible Fusion Framework	2-33
Object Properties	2-33
Object Functions	2-35
Example: Fuse Inertial Sensor Data Using insEKF	2-36
Occupancy Grids	2-45
Overview	2-45
World, Grid, and Local Coordinates	2-45
Inflation of Coordinates	2-46
Log-Odds Representation of Probability Values	2-50
Choose Path Planning Algorithms for Navigation	2-53
Execute Code at a Fixed-Rate	2-55
Introduction	2-55
Run Loop at Fixed Rate	2-55
Overrun Actions for Fixed Rate Execution	2-55
Particle Filter Workflow	2-58
Estimation Workflow	2-58
Particle Filter Parameters	2-62
Number of Particles	2-62
Initial Particle Location	2-63
State Transition Function	2-64
Measurement Likelihood Function	2-65
Resampling Policy	2-65
State Estimation Method	2-66
Pure Pursuit Controller	2-67
Reference Coordinate System	2-67
Look Ahead Distance	2-67
Limitations	2-68
Monte Carlo Localization Algorithm	2-69
Overview	2-69
State Representation	2-69
Initialization of Particles	2-71
Resampling Particles and Updating Pose	2-73
Motion and Sensor Model	2-74

Vector Field Histogram	2-78
Robot Dimensions	2-78
Cost Function Weights	2-80
Histogram Properties	2-80
Tune Parameters Using show	2-82

Navigation Block Examples

3

Convert Coordinate System Transformations	3-2
--	------------

nmeaParser Examples

4

Plot Position of GNSS Receiver Using Live NMEA Data or NMEA Log File	4-2
--	------------

Navigation Featured Examples

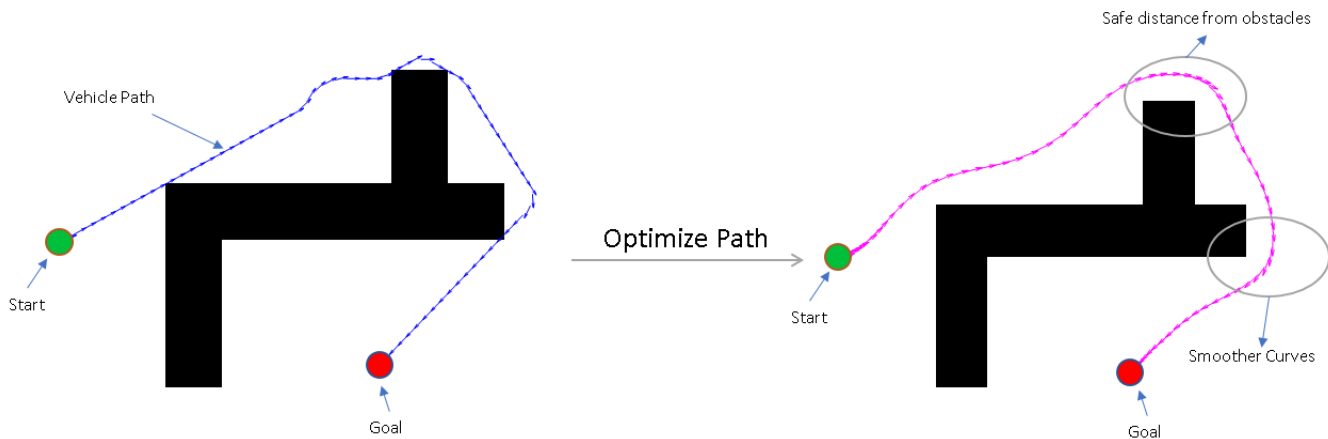
- “Optimization Based Path Smoothing for Autonomous Vehicles” on page 1-3
- “Benchmark Path Planners for Differential Drive Robots in Warehouse Map” on page 1-14
- “Generate Code for Path Planning Using Hybrid A Star” on page 1-30
- “Estimate Position and Orientation of a Ground Vehicle” on page 1-35
- “Pose Estimation From Asynchronous Sensors” on page 1-43
- “Inertial Sensor Noise Analysis Using Allan Variance” on page 1-48
- “Simulate Inertial Sensor Readings from a Driving Scenario” on page 1-59
- “Estimate Orientation Through Inertial Sensor Fusion” on page 1-67
- “IMU and GPS Fusion for Inertial Navigation” on page 1-77
- “GNSS Simulation Overview” on page 1-85
- “Estimate GNSS Receiver Position with Simulated Satellite Constellations” on page 1-93
- “Analyze GPS Satellite Visibility” on page 1-99
- “Simulate GPS Sensor Noise” on page 1-103
- “IMU Sensor Fusion with Simulink” on page 1-105
- “Automatic Tuning of the insfilterAsync Filter” on page 1-107
- “Custom Tuning of Fusion Filters” on page 1-115
- “Magnetometer Calibration” on page 1-125
- “Wheel Encoder Error Sources” on page 1-134
- “Remove Bias from Angular Velocity Measurement” on page 1-141
- “Detect Multipath GPS Reading Errors Using Residual Filtering in Inertial Sensor Fusion” on page 1-145
- “Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter” on page 1-151
- “Estimate Orientation with a Complementary Filter and IMU Data” on page 1-155
- “Estimate Orientation Using AHRS Filter and IMU Data in Simulink” on page 1-163
- “Estimate Phone Orientation Using Sensor Fusion” on page 1-172
- “Binaural Audio Rendering Using Head Tracking” on page 1-178
- “Estimating Orientation Using Inertial Sensor Fusion and MPU-9250” on page 1-183
- “Wireless Data Streaming and Sensor Fusion Using BNO055” on page 1-193
- “Rotations, Orientation, and Quaternions” on page 1-199
- “Lowpass Filter Orientation Using Quaternion SLERP” on page 1-214
- “Introduction to Simulating IMU Measurements” on page 1-218
- “Logged Sensor Data Alignment for Orientation Estimation” on page 1-230
- “Generate Off-Centered IMU Readings” on page 1-237
- “Estimate Robot Pose with Scan Matching” on page 1-242
- “Localize TurtleBot Using Monte Carlo Localization” on page 1-248

- “Compose a Series of Laser Scans with Pose Changes” on page 1-262
- “Minimize Search Range in Grid-based Lidar Scan Matching Using IMU” on page 1-266
- “Visual-Inertial Odometry Using Synthetic Data” on page 1-271
- “Landmark SLAM Using AprilTag Markers” on page 1-280
- “Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs” on page 1-292
- “Create Egocentric Occupancy Maps Using Range Sensors” on page 1-295
- “Build Occupancy Map from Lidar Scans and Poses” on page 1-300
- “Create Egocentric Occupancy Map from Driving Scenario Designer” on page 1-301
- “Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph” on page 1-307
- “Fuse Multiple Lidar Sensors Using Map Layers” on page 1-310
- “Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans” on page 1-319
- “Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans” on page 1-326
- “Perform SLAM Using 3-D Lidar Point Clouds” on page 1-333
- “Plan Mobile Robot Paths Using RRT” on page 1-341
- “Moving Furniture in a Cluttered Room with RRT” on page 1-347
- “Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map” on page 1-352
- “Motion Planning with RRT for a Robot Manipulator” on page 1-366
- “Dynamic Replanning on an Indoor Map” on page 1-372
- “Highway Lane Change” on page 1-378
- “Path Following with Obstacle Avoidance in Simulink®” on page 1-391
- “Obstacle Avoidance with TurtleBot and VFH” on page 1-395
- “Highway Trajectory Planning Using Frenet Reference Path” on page 1-397
- “Optimal Trajectory Generation for Urban Driving” on page 1-411
- “EKF-Based Landmark SLAM” on page 1-425
- “Reverse-Capable Motion Planning for Tractor-Trailer Model Using plannerControlRRT” on page 1-430
- “Simulate INS Block” on page 1-445
- “Object Tracking and Motion Planning Using Frenet Reference Path” on page 1-447
- “Offroad Planning with Digital Elevation Models” on page 1-458
- “Factor Graph-Based Pedestrian Localization with IMU and GPS Sensors” on page 1-478
- “Ground Vehicle Pose Estimation for Tightly Coupled IMU and GNSS” on page 1-483
- “Estimate Orientation Using GPS-Derived Yaw Measurements” on page 1-488
- “Design Fusion Filter for Custom Sensors” on page 1-494

Optimization Based Path Smoothing for Autonomous Vehicles

This example shows you how to optimize the path for a car-like robot by maintaining a smooth curvature and a safe distance from the obstacles in a parking lot.

In this example, you can use the `optimizePath` function along with the `optimizePathOptions` object to optimize a planned path. You can use any 2-D path planner like `plannerRRT`, `plannerRRTStar`, `plannerAstar`, `plannerHybridAStar`, etc. to plan a path from the entrance of the parking lot to a desired parking slot. In a parking lot like environment, the car often needs to take sharp turns and avoid obstacles like other cars, pillars, signboards, etc. The path generated by the path planners may not always be safe, easy to navigate, smooth, or kinematically feasible. In such situations path optimization becomes essential. The `optimizePathOptions` object has a large set of parameters and weights that you can tune to account for the environment and the vehicle constraints, the weights allows you to set their relative importance while optimizing the path.



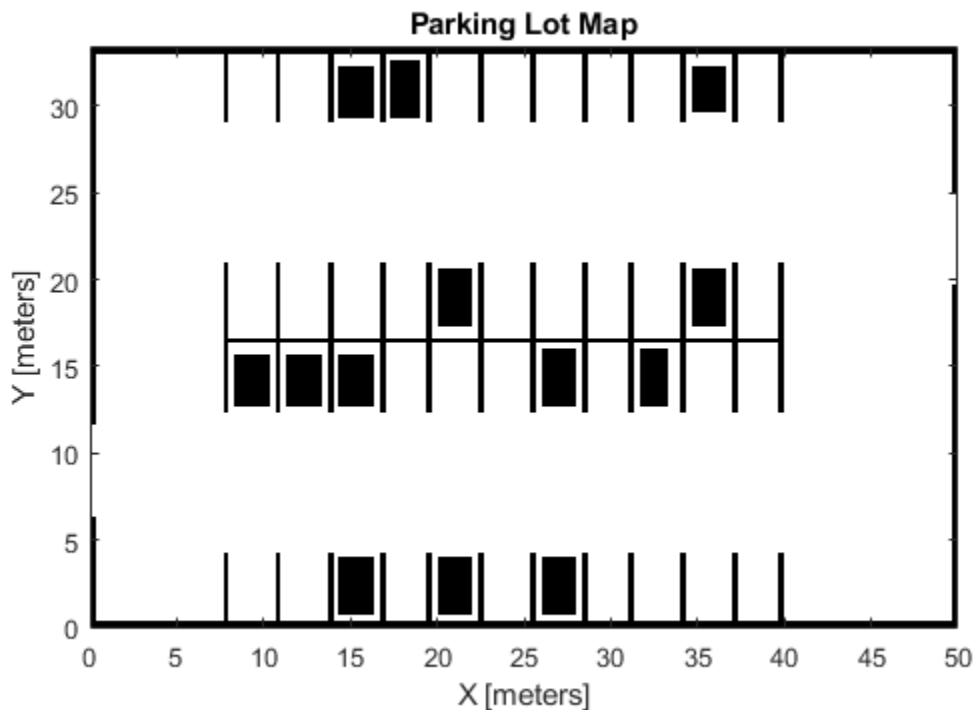
Set Up Parking Lot Environment

Create a `binaryOccupancyMap` object from a parking lot map and set the map resolution as 3 cells/meter.

```
load("parkingMap.mat");
resolution = 3;
map = binaryOccupancyMap(map, resolution);
```

Visualize the map. The map contains the floor plan of a parking lot with some of the parking slots already occupied.

```
show(map)
title("Parking Lot Map")
```



Choose Parking Spot

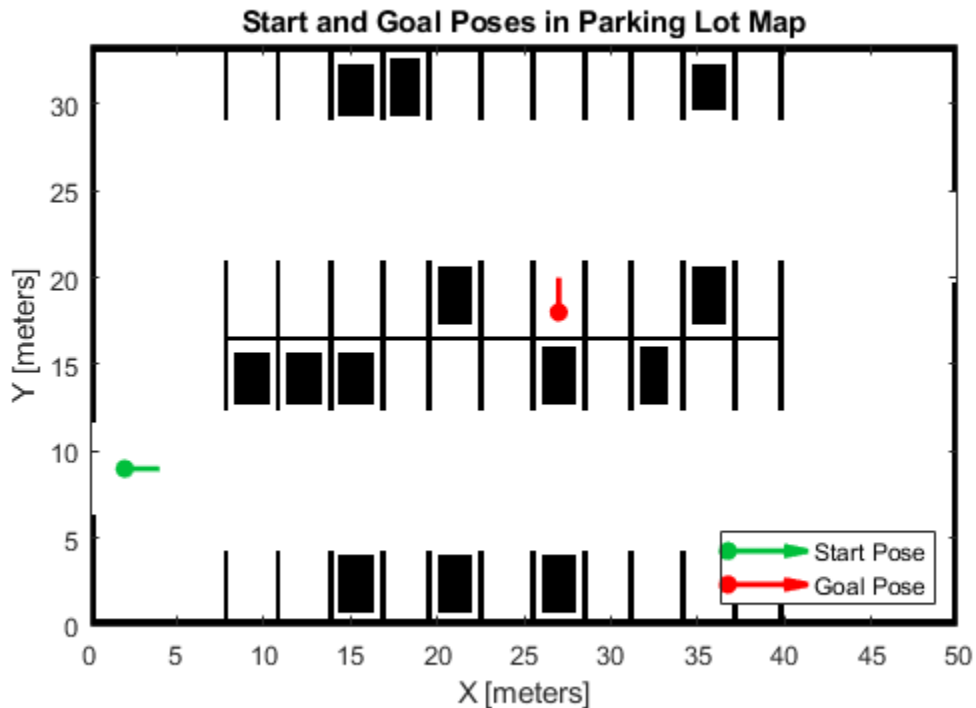
The current position of the car at the entrance of the parking lot is taken as the start pose and a desired unoccupied parking slot is chosen as the goal pose of the vehicle.

Define the start and goal poses for the vehicle as $[x \ y \ \theta]$ vectors. x and y specify the position in meters, and θ specifies the orientation angle in radians.

```
startPose = [2 9 0];
goalPose = [27 18 pi/2];
```

Visualize the poses.

```
show(map)
hold on
quiver(startPose(1),startPose(2),cos(startPose(3)),sin(startPose(3)),2,...
       color=[0 0.75 0.23],LineWidth=2,...
       Marker='o',MarkerFaceColor=[0 0.75 0.23],MarkerSize=5,...
       DisplayName="Start Pose",ShowArrowHead="off");
quiver(goalPose(1),goalPose(2),cos(goalPose(3)),sin(goalPose(3)),2, ...
       color=[1 0 0],LineWidth=2,...
       Marker='o',MarkerFaceColor=[1 0 0],MarkerSize=5,...
       DisplayName="Goal Pose",ShowArrowHead="off");
legend(Location="southeast");
title("Start and Goal Poses in Parking Lot Map")
hold off
```



Path Planning

Create a `validatorOccupancyMap` state validator using the `stateSpaceSE2` definition. Specify the map and the distance for interpolating and validating path segments.

```
validator = validatorOccupancyMap(stateSpaceSE2,map=map);
validator.ValidationDistance = 0.1;
```

Initialize the `plannerHybridAStar` object with the state validator object. Specify the `MinTurningRadius` and `MotionPrimitiveLength` properties of the planner.

```
planner = plannerHybridAStar(validator,MinTurningRadius=3,MotionPrimitiveLength=4);
```

Set default random number for repeatability.

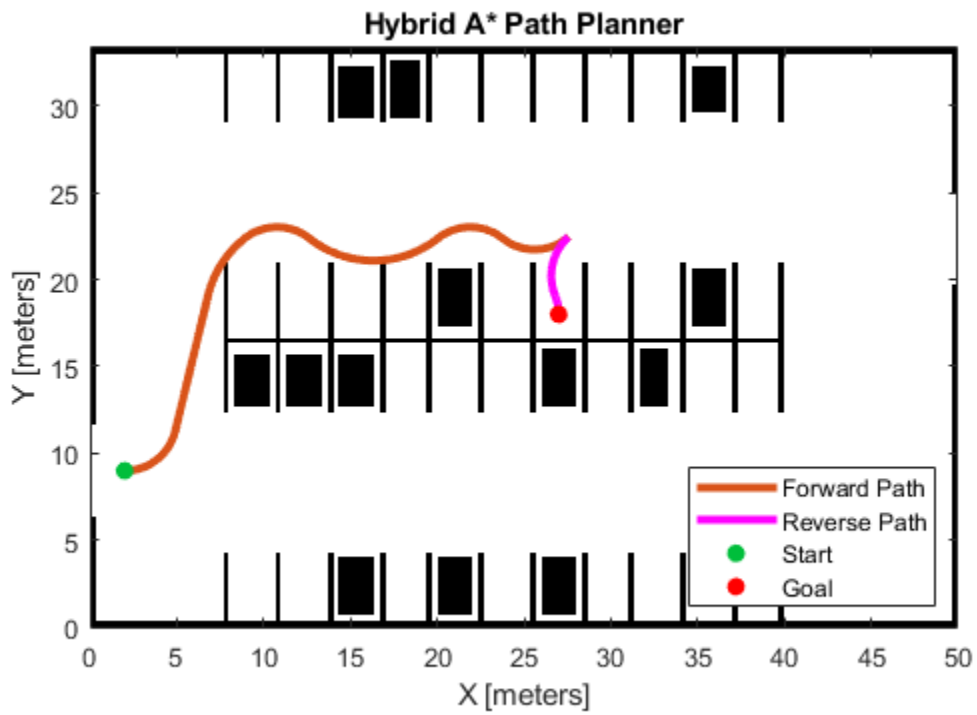
```
rng("default");
```

Plan a path from the start pose to the goal pose.

```
refPath = plan(planner,startPose,goalPose);
path = refPath.States;
```

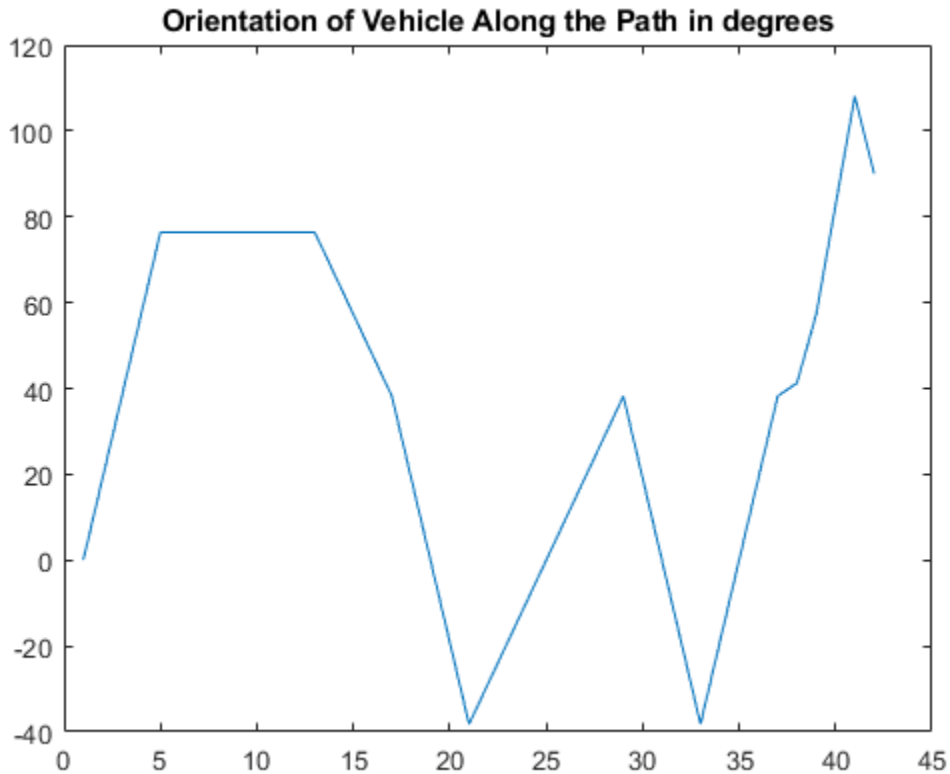
Visualize the planned path.

```
show(planner,Tree="off");
legend(Location="southeast");
hold off
```



Visualize the orientation of the vehicle.

```
plot(rad2deg(refPath.States(:,3)));  
title("Orientation of Vehicle Along the Path in degrees")
```



Configure Path Optimization Parameters

The path generated by the planner is composed of continuous path segments, but the junctions might be discontinuous. These junctions can lead to abrupt changes in the steering angle. The path may also contain segments that add extra driving time. To avoid such motion, the path needs to be optimized and smoothed. The path sometimes are very close to the obstacles, which can be risky, especially for heavy vehicles like trucks.

Create Optimization Options

Create `optimizePathOptions` object to configure the behaviour of the `optimizePath` function and the resulting path from it.

```
options = optimizePathOptions
```

```
options =  
optimizePathOptions
```

```
Trajectory Parameters  
  MaxPathStates: 200  
  ReferenceDeltaTime: 0.3000  
  MinTurningRadius: 1  
  MaxVelocity: 0.4000  
  MaxAngularVelocity: 0.3000  
  MaxAcceleration: 0.5000  
  MaxAngularAcceleration: 0.5000
```

```
Obstacle Parameters
    ObstacleSafetyMargin: 0.5000
    ObstacleCutOffDistance: 2.5000
    ObstacleInclusionDistance: 0.7500

Solver Parameters
    NumIteration: 4
    MaxSolverIteration: 15

Weights
    WeightTime: 10
    WeightSmoothness: 1000
    WeightMinTurningRadius: 10
    WeightVelocity: 100
    WeightAngularVelocity: 10
    WeightAcceleration: 10
    WeightAngularAcceleration: 10
    WeightObstacles: 50
```

Optimization options are grouped into four categories:

- 1 Trajectory Parameters
- 2 Obstacle Parameters
- 3 Solver Parameters
- 4 Weights

Trajectory Parameters

The trajectory parameters are used to specify the constraints of the vehicle while moving along the path like velocity, acceleration, turning radius, etc. They are soft limits which means that the solver might change them slightly while optimizing the path. Tune the following parameters,

- 1 **MinTurningRadius** - The turning radius of the vehicle. Increasing the **MinTurningRadius** will lead to larger path curvatures.
- 2 **MaxVelocity** - The maximum velocity that the vehicle can achieve. Changing this would modify the vehicle speed and trajectory time.
- 3 **MaxAcceleration** - The maximum acceleration possible for the vehicle.
- 4 **ReferenceDeltaTime** - Travel time between two consecutive poses. Increasing this would increase the vehicle speed.
- 5 **MaxPathStates** - Maximum number of poses allowed in path. Increasing this can give a smoother trajectory but might also increase the optimization time.

```
options.MinTurningRadius = 3; % meters
options.MaxVelocity = 5; % m/s
options.MaxAcceleration = 1; % m/s/s
options.ReferenceDeltaTime = 0.1; % second
```

```
separationBetweenStates = 0.2; % meters
numStates = refPath.pathLength/separationBetweenStates;
options.MaxPathStates = round(numStates);
```


Obstacle Parameters

The obstacle parameters specify the influence of the obstacles in the path. If there is a chance of the obstacles to move or their dimensions are not precisely known then you should keep the safety margin higher. In this example since most of the obstacles are lane markings and stationary parked vehicles, the safety margin can be smaller. Tune the following parameters,

- 1 **ObstacleSafetyMargin** - The safety margin that the path should maintain from the obstacles. Increasing the safety margin will make path safer and increase the distance from the obstacles.
- 2 **ObstacleInclusionDistance** - The obstacles within this distance from path will be included for the optimization.
- 3 **ObstacleCutOffDistance** - The obstacles beyond this distance from path will be ignored during optimization. This value must always be greater than obstacle inclusion distance.

For obstacles between the **ObstacleInclusionDistance** and **ObstacleCutOffDistance** the obstacles closest on the left and right sides between inclusion and cutoff distance are considered.

```
options.ObstacleSafetyMargin = 2; % meters
options.ObstacleInclusionDistance = 0.75; % meters
options.ObstacleCutOffDistance = 2.5; %i meters
```

Solver Parameters

The solver parameters specify the options for the solver used while optimizing the path. Higher values of these parameters improve the optimization results but also impact the optimization time, thus they should be tuned according to the need. Tune the following parameters,

- 1 **NumIteration** - Number of times solver is invoked during optimization, before each invocation the number of poses in the path are adjusted based on reference delta time.
- 2 **MaxSolverIteration** - Maximum number of iterations per solver invocation.

If path is already dense then the **NumIteration** can be reduced and **MaxSolverIteration** be increased. If path is sparse then **NumIteration** can be higher and **MaxSolverIteration** can be kept lower.

```
options.NumIteration = 4;
options.MaxSolverIteration = 15;
```

Weights

The weights define the relative importance of various constraints discussed above. Since most of the constraints are soft limits, the weights decide how important a constraint is while optimizing the path. Tune the following weights,

- 1 **WeightTime** - Weight of the time component, increasing this would lead to shorter travel time and path.
- 2 **WeightSmoothness** - Increasing this weight will make the path smoother.
- 3 **WeightMinTurningRadius** - Increasing this will try to maintain the turning radius greater than the minimum value for most of the path.
- 4 **WeightVelocity** - Increasing this will ensure that the velocity constraints are more closely followed.
- 5 **WeightObstacles** - Increasing this will ensure that vehicle does not cross an obstacle.

```
options.WeightTime = 10;
options.WeightSmoothness = 1000;
options.WeightMinTurningRadius = 10;
options.WeightVelocity = 10;
options.WeightObstacles = 50;
```

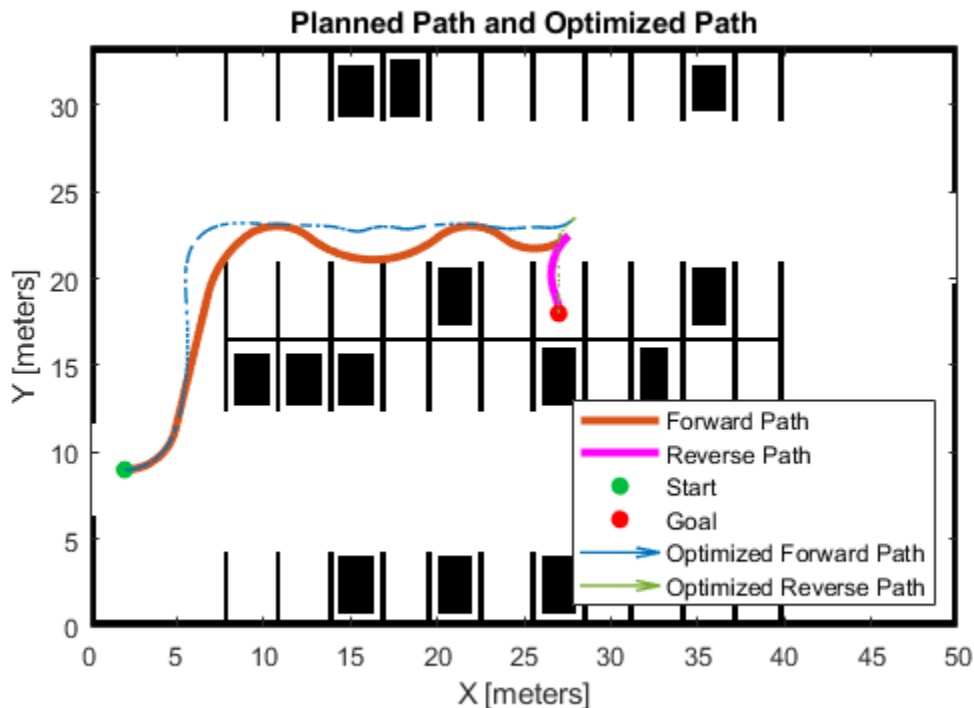
Path Optimization

Use the `optimizePath` function to optimize the path generated by the planner according to optimization options defined above.

```
[optimizedPath,kineticInfo] = optimizePath(path,map,options);
drivingDir = sign(kineticInfo.Velocity);
```

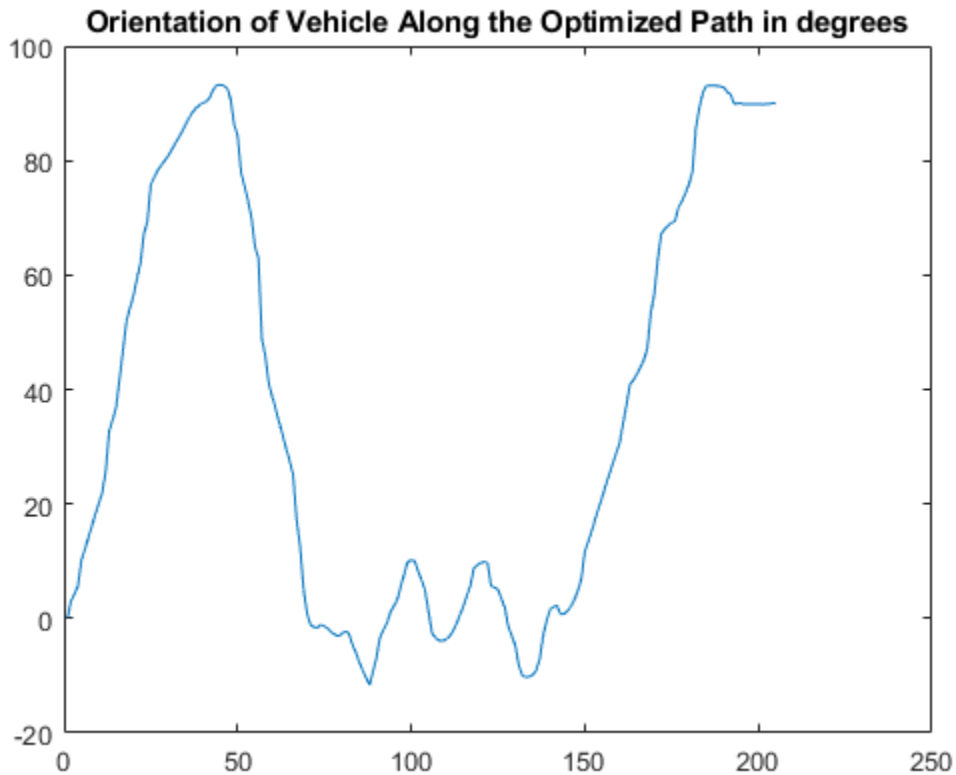
Visualize the optimized path.

```
show(planner,Tree="off");
hold on
forwardMotion = optimizedPath(drivingDir==1,:);
reverseMotion = optimizedPath(drivingDir==-1,:);
quiver(forwardMotion(:,1),forwardMotion(:,2),cos(forwardMotion(:,3)),sin(forwardMotion(:,3)),...
    0.1,Color=[0 0.45 0.74],LineWidth=1,DisplayName="Optimized Forward Path");
quiver(reverseMotion(:,1),reverseMotion(:,2),cos(reverseMotion(:,3)),sin(reverseMotion(:,3)),...
    0.1,Color=[0.47 0.68 0.19],LineWidth=1,DisplayName="Optimized Reverse Path");
legend(Location="southeast");
title("Planned Path and Optimized Path")
hold off
```



Plot the orientation of the vehicle along the optimized path.

```
plot(rad2deg(optimizedPath(:,3)))
title("Orientation of Vehicle Along the Optimized Path in degrees")
```



Tune Parameters with Live Controls

In the above section the path was optimized based on some preset parameter values. However, setting a good set of parameters can have a significant impact on the path.

In this section you can update the parameters using the sliders and visualize the impact they have on the optimized path. This will help you get a better understanding on how the parameters impact the final smooth path.

Trajectory Parameters

options.MinTurningRadius = 1.5 ; % m

options.MaxVelocity = 5 ; % m/s

options.MaxAcceleration = 1 ; % m/s/s




ReferenceDeltaTime is an important parameter and can have huge impact on the results.

options.ReferenceDeltaTime = 0.2 ; % s



separationBetweenStates = 0.2 ; % m

```
numStates = refPath.pathLength/separationBetweenStates;
options.MaxPathStates = round(numStates);
```






Obstacle Parameters

```
options.ObstacleSafetyMargin = 1.5  ; % m
options.ObstacleCutOffDistance = 4  ; % m
options.ObstacleInclusionDistance = 2  ; % m
```

Solver Parameters

```
options.NumIteration = 2  ;
options.MaxSolverIteration = 8  ;
```

Weights

```
options.WeightTime = 200  ;
options.WeightSmoothness = 1000  ;
options.WeightMinTurningRadius = 140  ;
options.WeightVelocity = 230  ;
options.WeightObstacles = 320  ;
```

Optimize Path and Visualize

The new optimized path is plotted along with the previously generated and optimized path. This will help you compare the optimization results.

First plot the original path planned by the planner and the results of the previous optimization.

```
show(planner,Tree="off")
hold on
quiver(forwardMotion(:,1),forwardMotion(:,2),cos(forwardMotion(:,3)),sin(forwardMotion(:,3)),...
    0.1,Color=[0 0.45 0.74],LineWidth=1,DisplayName="Previous Optimized Forward Path");
quiver(reverseMotion(:,1),reverseMotion(:,2),cos(reverseMotion(:,3)),sin(reverseMotion(:,3)),...
    0.1,Color=[0.47 0.68 0.19],LineWidth=1,DisplayName="Previous Optimized Reverse Path");
```

Now optimize the path based on the new set of optimization options.

```
[optimizedPath,kineticInfo] = optimizePath(path,map,options);
```

Finally plot the new optimized path.

```
drivingDir = sign(kineticInfo.Velocity);

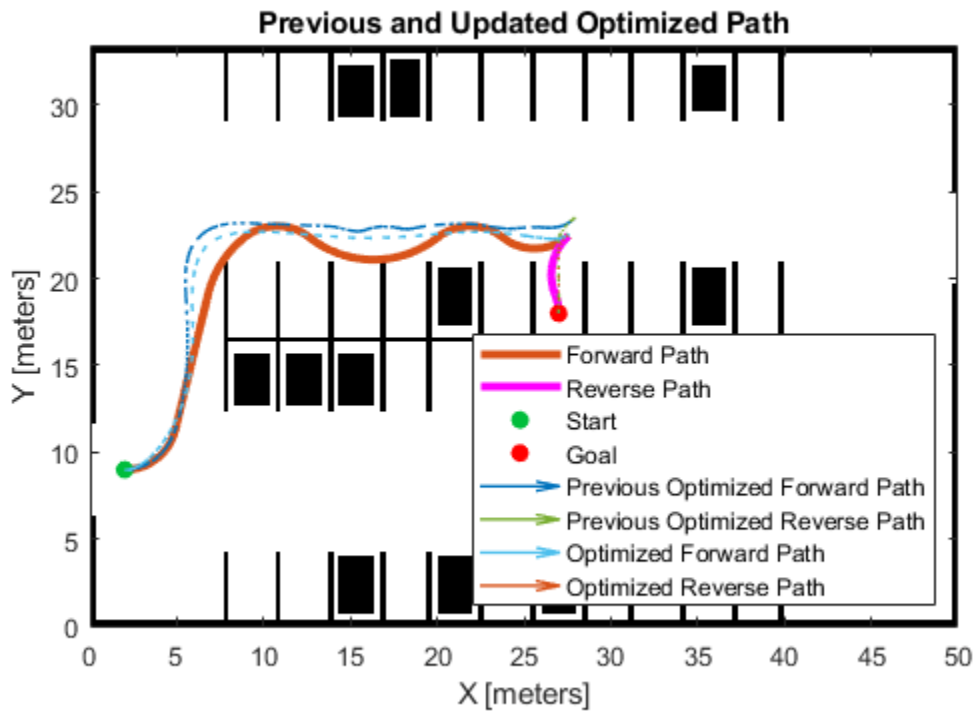
forwardMotion = optimizedPath(drivingDir==1,:);
reverseMotion = optimizedPath(drivingDir==-1,:);

quiver(forwardMotion(:,1),forwardMotion(:,2),cos(forwardMotion(:,3)),sin(forwardMotion(:,3)),...
    0.1,Color=[0.3 0.75 0.93],LineWidth=1,DisplayName="Optimized Forward Path");
quiver(reverseMotion(:,1),reverseMotion(:,2),cos(reverseMotion(:,3)),sin(reverseMotion(:,3)),...
    0.1,Color=[0.47 0.68 0.19],LineWidth=1,DisplayName="Optimized Reverse Path");
```

```

0.1,Color=[0.85 0.33 0.1],LineWidth=1,DisplayName="Optimized Reverse Path");
legend(Location="southeast");
title("Previous and Updated Optimized Path")
hold off

```



Reference

Rosmann, Christoph, Frank Hoffmann, and Torsten Bertram. "Kinodynamic Trajectory Optimization and Control for Car-like Robots." In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 5681-86. Vancouver, BC: IEEE, 2017. <https://doi.org/10.1109/IROS.2017.8206458>.

Benchmark Path Planners for Differential Drive Robots in Warehouse Map

This example shows how to choose the best 2-D path planner for a differential drive robot in a warehouse environment from the available path planners. Use the `plannerBenchmark` object to benchmark the path planners `plannerRRT`, `plannerRRTStar`, `plannerBiRRT`, `plannerPRM`, and `plannerHybridAstar` on the warehouse environment with the randomly chosen start and goal poses. Compare the path planners based on their ability to find a valid path, clearance from the obstacles, time taken to initialize a planner, time taken to find a path, length of the path, and smoothness of the path. A suitable planner is chosen based on the performance of each path planner on the above mentioned metrics.

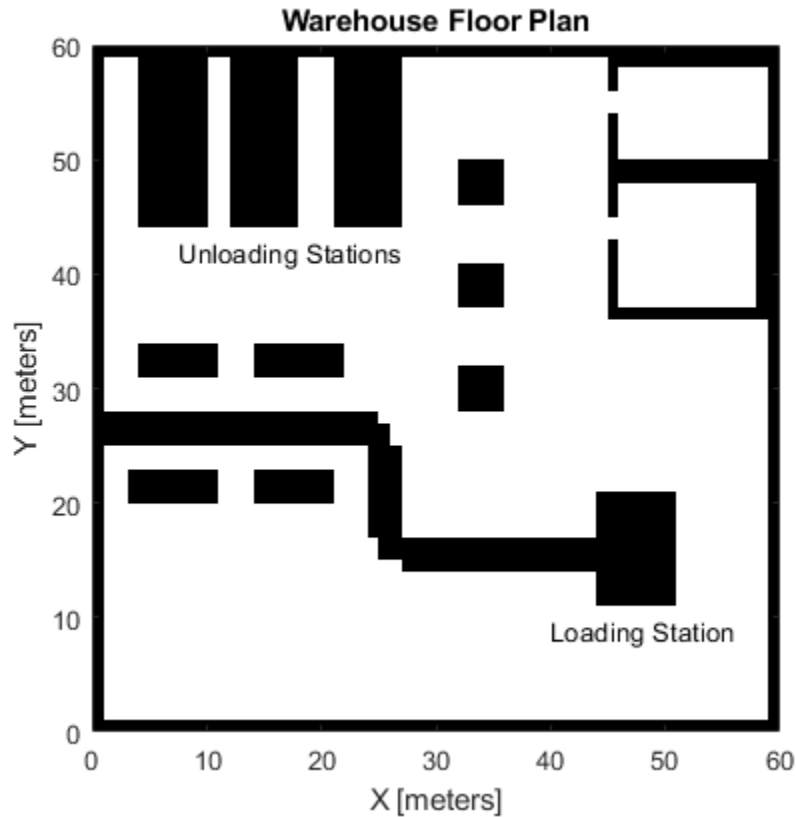
Set Up Environment

Create a `binaryOccupancyMap` object from a warehouse map. In the warehouse, the mobile robot moves from the loading station to the unloading station to place the goods.

```
map = load("warehouseMap.mat").logicalMap;  
map = binaryOccupancyMap(map);
```

Visualize the map.

```
figure  
show(map)  
title("Warehouse Floor Plan")  
% Set the location of text displayed on the map.  
loadingStationTextLoc = [40 9 0];  
unloadingStationTextLoc = [7.5 42 0];  
hold on  
text(loadingStationTextLoc(1),loadingStationTextLoc(2),1,"Loading Station");  
text(unloadingStationTextLoc(1),unloadingStationTextLoc(2),1,"Unloading Stations");  
hold off
```



Define Environment and Planners for Benchmarking

Specify the minimum turning radius for the differential drive robot.

```
minTurningRadius = 2.2; % meters
```

Create a `stateSpaceDubins` object with the state space bounds to be the same as map limits. set the minimum turning radius.

```
stateSpace = stateSpaceDubins([map.XWorldLimits; map.YWorldLimits; [-pi pi]]);
stateSpace.MinTurningRadius = minTurningRadius;
```

Create a `validatorOccupancyMap` state validator with the Dubins state space using the map. Specify the distance for interpolating and validating path segments.

```
validator = validatorOccupancyMap(stateSpace, Map=map);
validator.ValidationDistance = 0.01*(1/map.Resolution); % meters
```

Define the function handles for the initialization functions of each planners. For more information on these initialization functions, see Initialization Functions for Planners on page 1-0 .

```
rrtInit = @(validator) plannerRRTWrapper(validator);
rrtStarInit = @(validator) plannerRRTStarWrapper(validator);
birrtInit = @(validator) plannerBiRRTWrapper(validator);
haStarInit = @(validator) plannerHybridAStarWrapper(validator, minTurningRadius);
prmInit = @(validator) plannerPRM(validator.StateSpace, validator);
```

Define the function handle for the plan function, which is common for all the planners.

```
planFcn = @(initOutput,start,goal) plan(initOutput,start,goal);
```

Randomly Select Start-Goal Pairs on Warehouse Map

The start and goal locations are randomly sampled from the loading and unloading station area, respectively. Specify the number of start-goal pairs that must be randomly generated. In this example only three start-goal pair are chosen to reduce the execution time of this example. Increase the start-goal pair number to get sufficient map coverage.

```
% Set default random number for repeatability of results.
rng("default")
% Select the number of start-goal pairs.
numStartGoalPairs = 3;
```

The start location of the robot is sampled from the rectangle area marked as loading station and goal location is sampled from the rectangle area marked as unloading station area. Robot locations are sampled uniformly in the marked area. The rectangle area is specified as a vector of form $[x \ y \ w \ h]$. x and y specify the coordinate of the bottom left corner of the rectangle. w and h specify the width and height of the rectangle.

```
loadingArea = [51.5 11 5 10];
startLocations = helperSampleSelectedAreaOnMap(validator,loadingArea,numStartGoalPairs);

unloadingArea = [2 43.5 27 15];
goalLocations = helperSampleSelectedAreaOnMap(validator,unloadingArea,numStartGoalPairs);
```

Visualize the map with all the randomly sampled start and goal locations.

```
show(map)
title("Warehouse Floor Plan")
% Indicate the loading and unloading station on the map.
hold on
text(loadingStationTextLoc(1),loadingStationTextLoc(2),1,"Loading Station");
rectangle(Position=loadingArea)

text(unloadingStationTextLoc(1),unloadingStationTextLoc(2),1,"Unloading Stations");
rectangle(Position=unloadingArea)

% Set the length of the line representing the pose in the start-goal visualization.
r = 2;

% Display all the start-goal pairs on the map.
for i=1:numStartGoalPairs
    start = startLocations(i,:);
    goal = goalLocations(i,:);

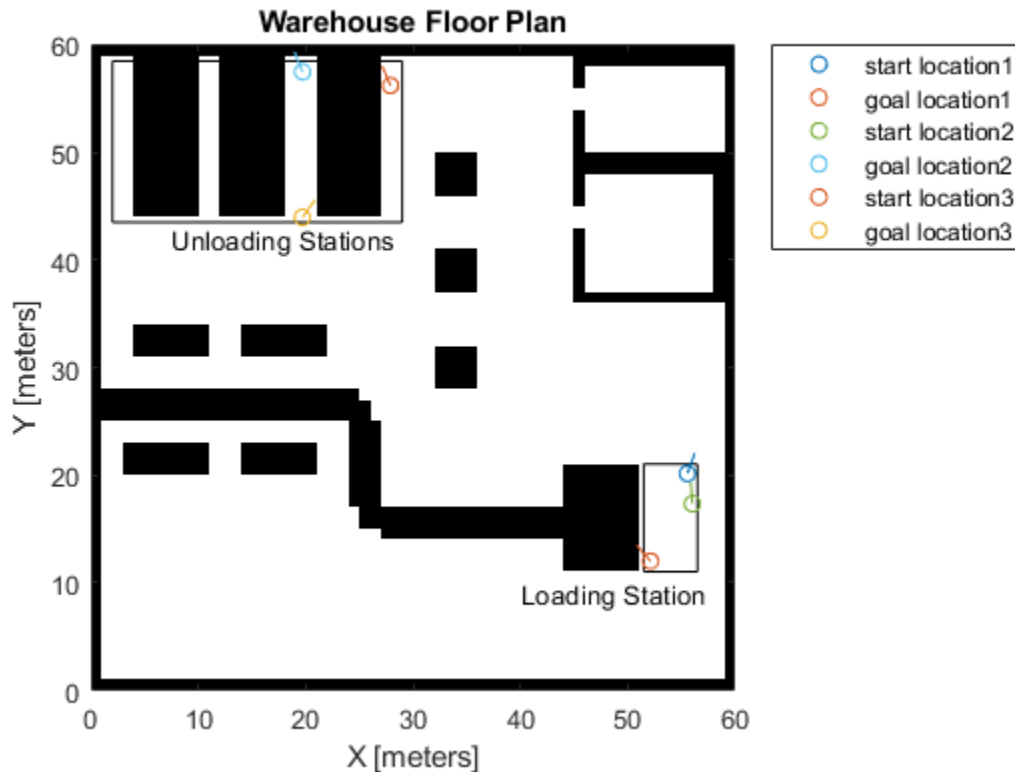
    % Define the legend displayed on the map
    startString = strcat("start location",num2str(i));
    goalString = strcat("goal location",num2str(i));
    % Display start and goal location of robot.
    p1 = plot(start(1,1),start(1,2),"o",DisplayName=startString);
    c1 = p1.Color;
    p2 = plot(goal(1,1),goal(1,2),"o",DisplayName=goalString);
    c2 = p2.Color;
    % Display start and goal headings.
    plot([start(1) start(1)+r*cos(start(3))],[start(2) start(2)+r*sin(start(3))],...
        "-",Color=c1,HandleVisibility="off")
    plot([goal(1) goal(1)+r*cos(goal(3))],[goal(2) goal(2)+r*sin(goal(3))],...
        "-",Color=c2,HandleVisibility="off")
end
```



```

        "-",Color=c2,HandleVisibility="off")
end
hold off
legend(Location="northeastoutside")

```



Planner Benchmark Object Creation and Running Benchmark

Create a `plannerBenchmark` object for each start-goal pair and add the planners for benchmarking. The planners are executed twice on the same environment and start-goal pair by setting the `runCount` value to 2. This ensures the metrics results are accurate since sampling based planners like `plannerRRT`, `plannerRRTStar`, `plannerBiRRT`, and `plannerPRM` produce different output on the same set of start-goal pair. In this example `runCount` value is set to 2 to reduce the execution time of this example. Increase the `runCount` value to get more accurate benchmark results.

```

% To store the generated benchmark objects for each start-goal pair.
benchmarkList = cell(1,numStartGoalPairs);
% Specify the number of times each planner to be executed on the same
% set of start-goal pair.
runCount = 2;

for i=1:size(startLocations,1)
    % Get each start and goal location from all the sampled locations.
    start = startLocations(i,:);
    goal = goalLocations(i,:);

    % Set default random number for repeatability of results.

```

```
rng("default")
% Construct benchmark object.
benchmark = plannerBenchmark(validator,start,goal);
% Add planners for benchmarking using initialization and plan function
% handles. Additional optional input NumPlanOutput define the number of
% outputs returned from the plan function.
addPlanner(benchmark,planFcn,rrtInit,PlannerName="rrt",NumPlanOutput=2)
addPlanner(benchmark,planFcn,rrtStarInit,PlannerName="rrtStar",NumPlanOutput=2)
addPlanner(benchmark,planFcn,birrtInit,PlannerName="biRRT",NumPlanOutput=2)
addPlanner(benchmark,planFcn,prmInit,PlannerName="plannerPRM",NumPlanOutput=2)
addPlanner(benchmark,planFcn,haStarInit,PlannerName="hybridAstar",NumPlanOutput=2)
% Run the benchmark.
runPlanner(benchmark,runCount)
% Store the benchmark for further analysis.
benchmarkList{i} = benchmark;
end

Initializing rrt ...
Done.
Planning with rrt for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Planning with rrt for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Initializing rrtStar ...
Done.
Planning with rrtStar for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Planning with rrtStar for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Initializing biRRT ...
Done.
Planning with biRRT for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Planning with biRRT for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Initializing plannerPRM ...
Done.
Planning with plannerPRM for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Planning with plannerPRM for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Initializing hybridAstar ...
Done.
Planning with hybridAstar for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Planning with hybridAstar for start pose (55.5736 20.1338 1.2229) and goal pose (27.9063 56.2369 1.9757)
Initializing rrt ...
Done.
Planning with rrt for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Planning with rrt for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Initializing rrtStar ...
Done.
Planning with rrtStar for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Planning with rrtStar for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Initializing biRRT ...
Done.
Planning with biRRT for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Planning with biRRT for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Initializing plannerPRM ...
Done.
Planning with plannerPRM for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Planning with plannerPRM for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Initializing hybridAstar ...
Done.
Planning with hybridAstar for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Planning with hybridAstar for start pose (56.029 17.3236 1.6444) and goal pose (19.705 57.5099 1.9527) at
Initializing rrt ...
```

```

Done.
Planning with rrt for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Planning with rrt for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Initializing rrtStar ...
Done.
Planning with rrtStar for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Planning with rrtStar for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Initializing biRRT ...
Done.
Planning with biRRT for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Planning with biRRT for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Initializing plannerPRM ...
Done.
Planning with plannerPRM for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Planning with plannerPRM for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Initializing hybridAstar ...
Done.
Planning with hybridAstar for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)
Planning with hybridAstar for start pose (52.1349 11.9754 2.2894) and goal pose (19.6979 43.9775 0.93797)

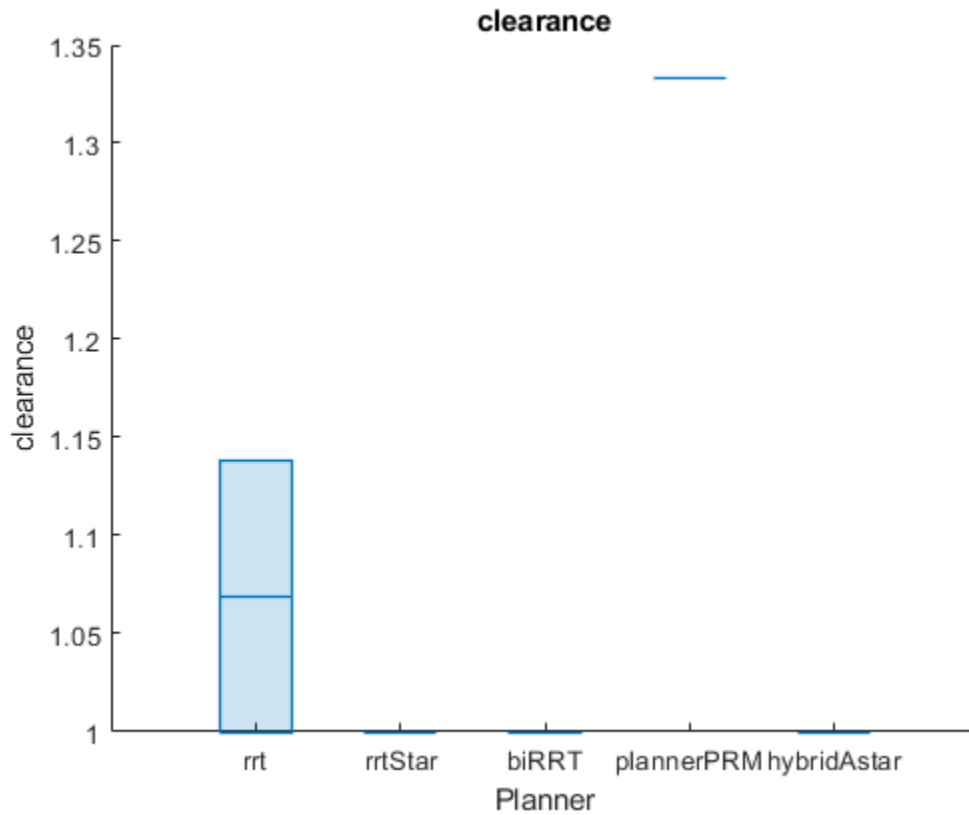
```

Average Metrics Across all Start-Goal Pairs

All the planner are executed `runCount` times for each start-goal pair. Also all the planners are executed for all the start-goal pairs. This means that all planners are executed `runCount*numStartGoalPairs` times. The plots and tables below show the average metric value across all the start-goal pairs. The tables represent the Mean, Median, and Standard Deviation of each metric averaged across all the start-goal pairs.

The `clearance` metric represent the minimum distance of the path from the obstacles in the environment. The plot shows that `plannerPRM` has the highest clearance.

```
helperPlotAveragedMetrics(benchmarkList,runCount,"clearance")
```



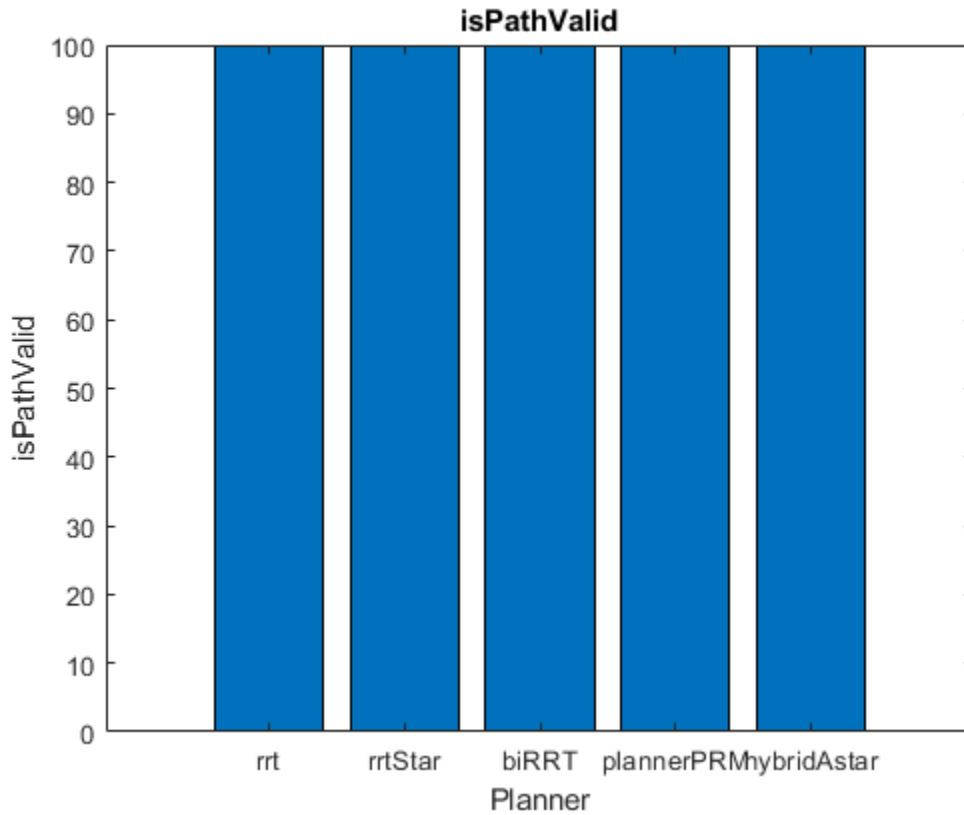
```
clearanceAverage = helperCalculateAverageMetricTable(benchmarkList, "clearance")
```

```
clearanceAverage=5x3 table
```

	Mean	Median	stdDev
rrt	1.069	1	0.097631
rrtStar	1	1	0
biRRT	1	1	0
plannerPRM	1.3333	1	0.4714
hybridAstar	1	1	0

The `isPathValid` metric represent the success rate of each planner expressed in percentage. The plot shows that all the path planners produced valid path for all the start-goal pairs.

```
helperPlotAveragedMetrics(benchmarkList, runCount, "isPathValid")
```

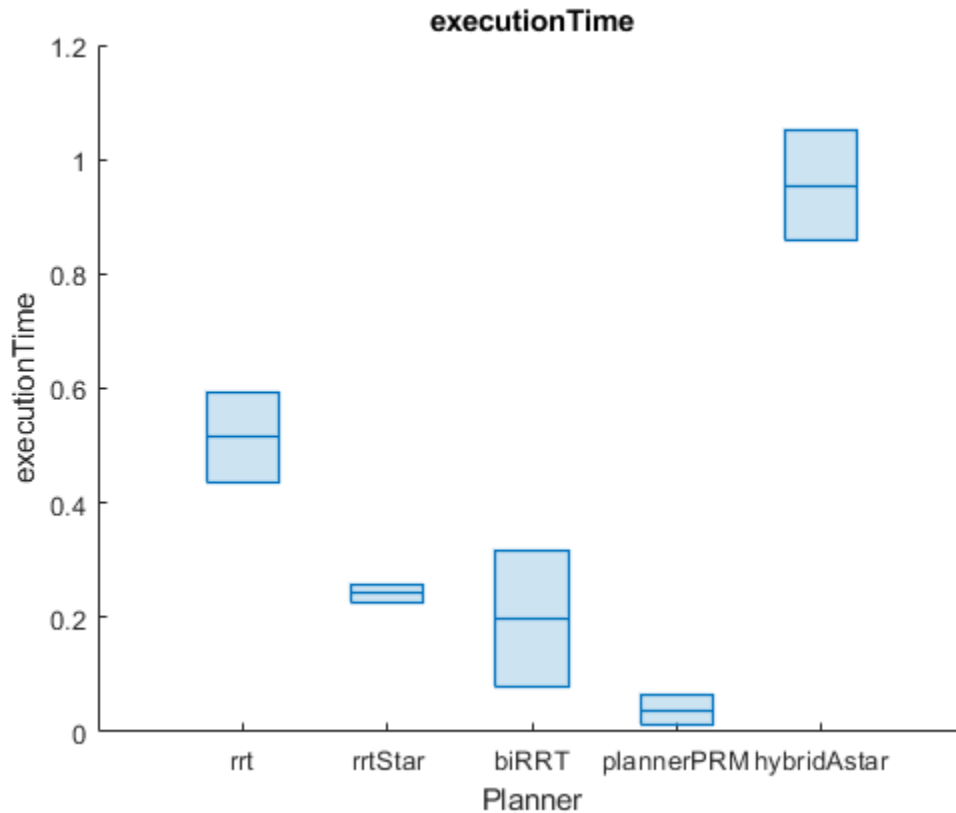


```
isPathValidAverage = helperCalculateAverageMetricTable(benchmarkList, "isPathValid")
```

```
isPathValidAverage=5x3 table
      Mean   Median   stdDev
-----
rrt      100     100      0
rrtStar  100     100      0
biRRT    100     100      0
plannerPRM 100     100      0
hybridAstar 100     100      0
```

The `executionTime` metric represent the time taken by the plan function to execute. The plot shows that `plannerPRM` took the least time, followed by `plannerBiRRT`. We could also note that `plannerRRTStar` took lesser time than `plannerRRT`, this could be due to the lesser number of start-goal pairs used for benchmarking.

```
helperPlotAveragedMetrics(benchmarkList, runCount, "executionTime")
```



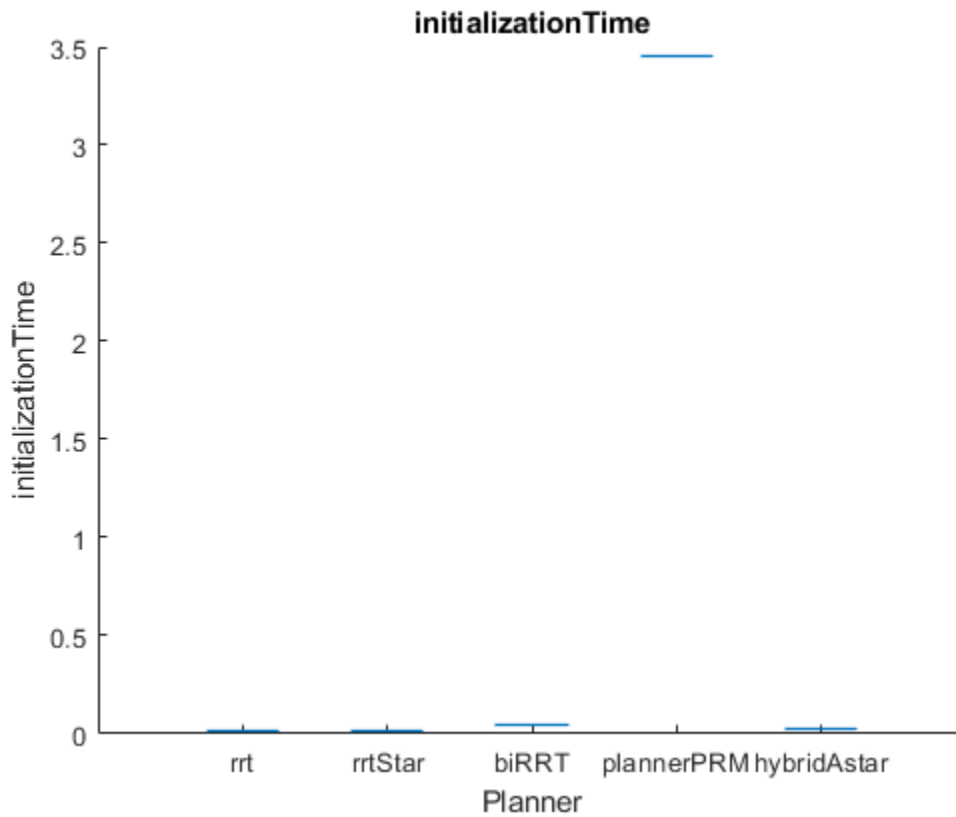
```
execTimeAverage = helperCalculateAverageMetricTable(benchmarkList, "executionTime")
```

```
execTimeAverage=5x3 table
```

	Mean	Median	stdDev
rrt	0.51461	0.47376	0.11831
rrtStar	0.24254	0.21971	0.046682
biRRT	0.19747	0.0977	0.15746
plannerPRM	0.038455	0.026413	0.019184
hybridAstar	0.95421	1.1552	0.47027

The `initializationTime` metric indicates the time taken to execute the initialization function of each planner. Hence total execution time is the sum of plan function execution time and initialization time. `plannerPRM` has the longest initialization time. Hence `plannerBiRRT` took the least total execution time.

```
helperPlotAveragedMetrics(benchmarkList, runCount, "initializationTime")
```



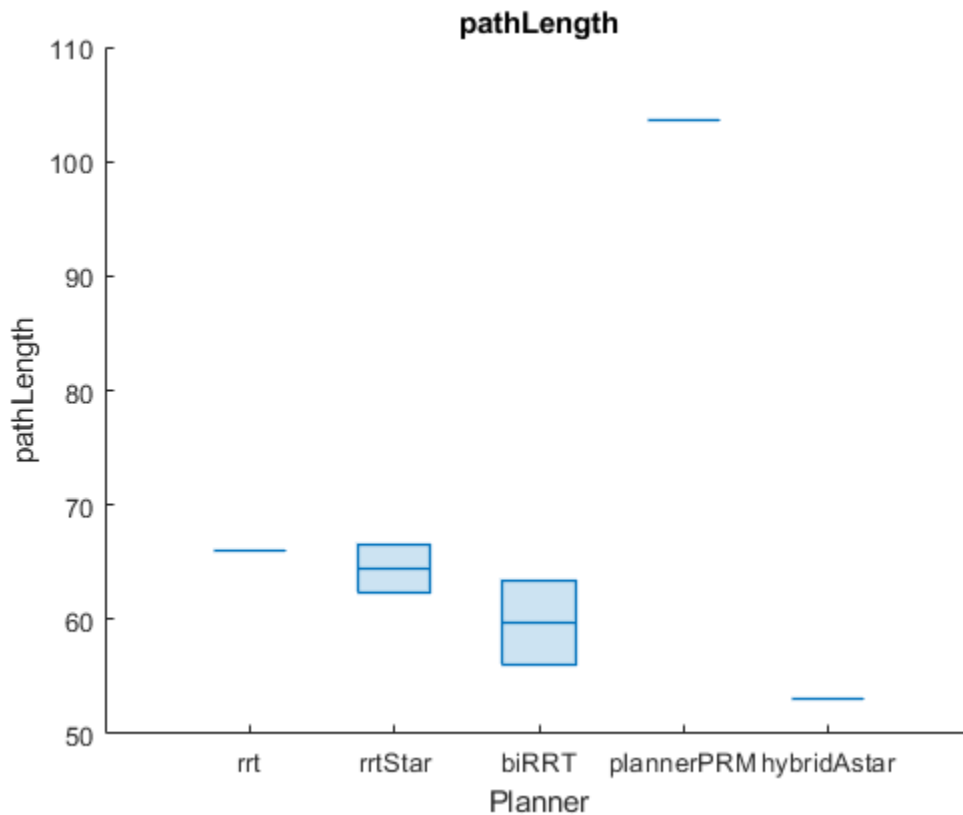
```
initTimeAverage = helperCalculateAverageMetricTable(benchmarkList, "initializationTime")
```

```
initTimeAverage=5x3 table
```

	Mean	Median	stdDev
rrt	0.014326	0.011862	0.0037965
rrtStar	0.01037	0.011904	0.0045337
biRRT	0.048969	0.014486	0.053037
plannerPRM	3.4556	3.1519	0.50742
hybridAstar	0.022985	0.026037	0.0095362

The pathLength metric represent the length of the generated path. plannerHybridAstar has the shortest path, followed by plannerBiRRT.

```
helperPlotAveragedMetrics(benchmarkList, runCount, "pathLength")
```



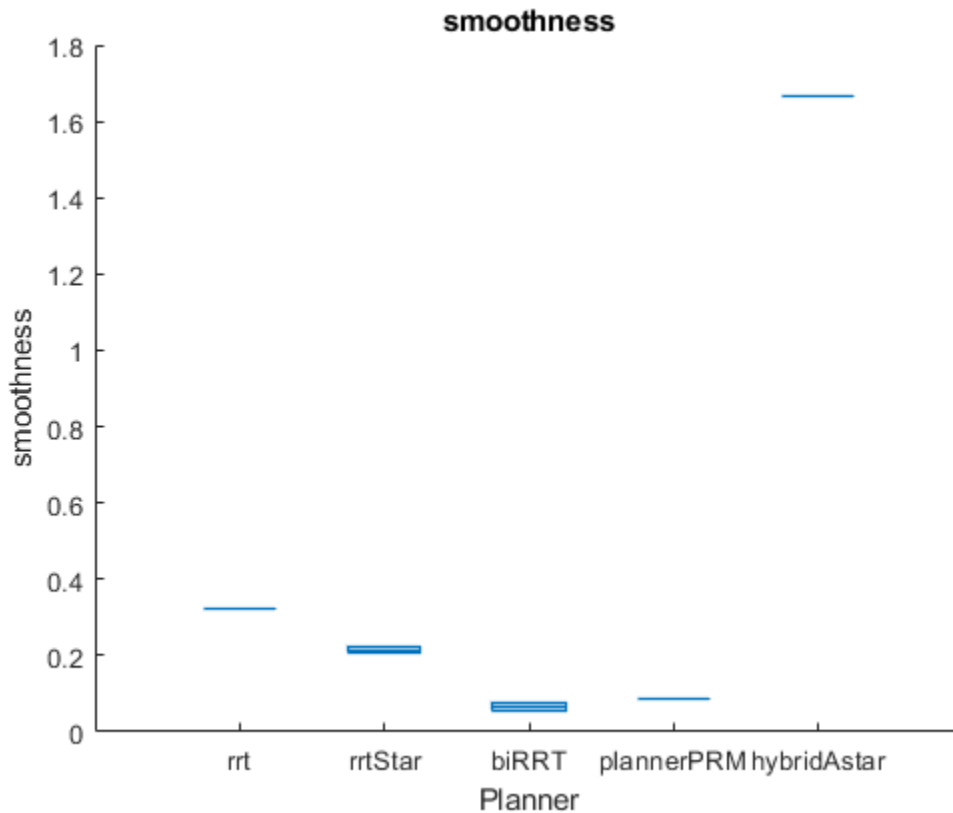
```
pathLengthAverage = helperCalculateAverageMetricTable(benchmarkList, "pathLength")
```

```
pathLengthAverage=5x3 table
```

	Mean	Median	stdDev
rrt	65.992	69.037	7.3361
rrtStar	64.426	67.992	5.3652
biRRT	59.68	63.763	8.4716
plannerPRM	103.67	103.34	1.3802
hybridAstar	53.038	50.676	4.7331

The smoothness metric represent the smoothness of the path for all poses. `plannerBiRRT` produced the smoothest path (lower smoothness value indicate smoother path) followed by `plannerPRM`. Observe that `plannerRRTStar` produced considerably smoother path compared to `plannerRRT`.

```
helperPlotAveragedMetrics(benchmarkList, runCount, "smoothness")
```

```
smoothnessAverage = helperCalculateAverageMetricTable(benchmarkList, "smoothness")
```

```
smoothnessAverage=5x3 table
```

	Mean	Median	stdDev
rrt	0.32247	0.31304	0.054903
rrtStar	0.21445	0.23351	0.038925
biRRT	0.064741	0.029512	0.069532
plannerPRM	0.086357	0.097025	0.025469
hybridAstar	1.6664	1.647	0.16529

If the slightly longer path is not a concern then consider using `plannerBiRRT` for path planning in a warehouse map setup as shown in this example, since it has the smoothest path and lesser execution time. If the path travelled by the robot should be the least then `plannerHybridAstar` could be considered.

As it can be seen, the inconsistency in execution time like, `plannerRRTStar` taking lesser time than `plannerRRT`, increasing the number of start-goal pairs and `runCount` to a considerably larger value will produce more accurate metrics results and better judgement of which path planner to choose.

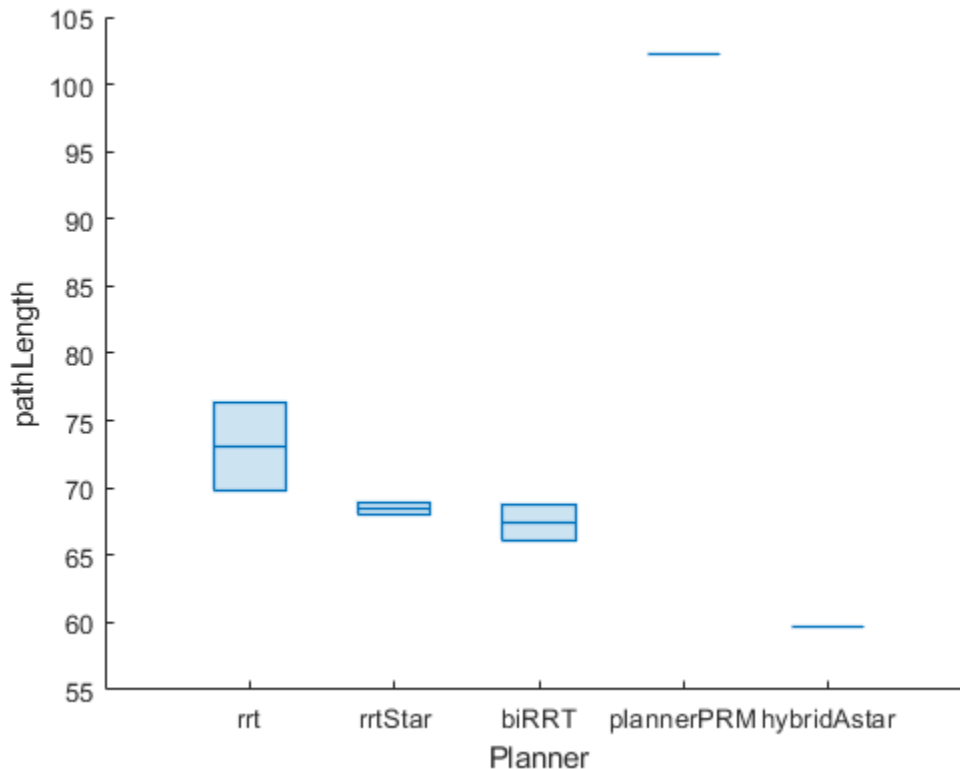
Visualize Metric Results for Specific Start-Goal Pair

The above tables and graphs showed the metric results averaged across all the start-goal pairs. `plannerBiRRT` produced smoother path overall but the path length was slightly longer than

`plannerHybridAstar`, which produced the least path length. Probe the start-goal pairs to see which produced the longest path length for `plannerBiRRT`.

```
pathLengthMean = zeros(1,numStartGoalPairs);
% Iterate over all the benchmarks and store the mean path length for each
% start-goal pair.
for i=1:numStartGoalPairs
    benchmark = benchmarkList{i};
    pathLengthTable= benchmark.metric("pathLength");
    pathLengthMean(i) = pathLengthTable.Mean("biRRT");
end
% Find the index of the benchmark which produced largest mean path length
% value for plannerBiRRT.
[~,largestPathLengthIdx] = max(pathLengthMean);
benchmarkLargestPathlength = benchmarkList{largestPathLengthIdx};

show(benchmarkLargestPathlength,"pathLength")
```



```
pathLength = metric(benchmarkLargestPathlength,"pathLength")
```

```
pathLength=5x4 table
```

	Mean	Median	StdDev	sampleSize
rrt	73.058	73.058	3.3079	2
rrtStar	68.444	68.444	0.49012	2
biRRT	67.393	67.393	1.3109	2
plannerPRM	102.18	102.18	0	2

```
hybridAstar    59.643    59.643         0         2
```

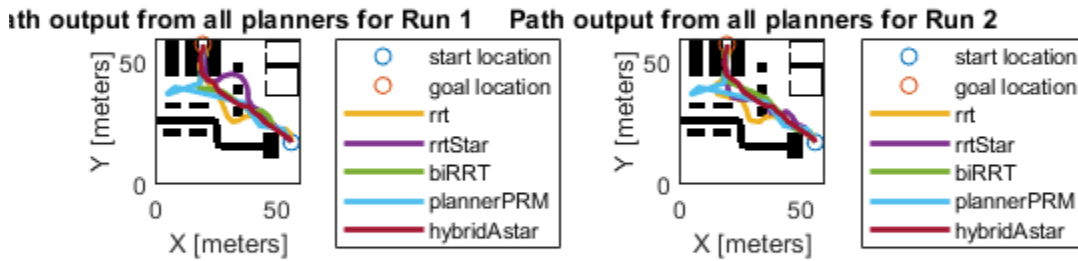
Visualize Path From All Planners for Specific Start-Goal Pair

Visualize the path output from all the planners for the start-goal pair that produced the longest path length for plannerBiRRT.

```
% Retrieve the start and goal location from the benchmark object.
start = benchmarkLargestPathlength.Start;
goal = benchmarkLargestPathlength.Goal;

tiledlayout("flow")
% Display the path from start location to goal location for all the path
% planners for all the runs in runCount.
for run=1:runCount
    nexttile
    show(map)
    title(['Path output from all planners for Run ' num2str(run)])
    hold on

    % show start and goal positions of robot.
    plot(start(1,1),start(1,2),"o")
    plot(goal(1,1),goal(1,2),"o")
    % Find the planner names used for benchmarking.
    plannerNames = fieldnames(benchmarkLargestPathlength.PlannerOutput);
    numPlanners = length(plannerNames);
    runString = strcat("Run",num2str(run));
    % Iterate and plot path of each planner for the specified run.
    for i=1:numPlanners
        plannerName = plannerNames{i};
        plannerOutput = benchmarkLargestPathlength.PlannerOutput.(plannerName).PlanOutput.(runString);
        pathObj = plannerOutput{1};
        plot(pathObj.States(:,1),pathObj.States(:,2),"-", "LineWidth",2)
    end
end
% Specify the legends.
labels = [{"start location","goal location"} plannerNames'];
legend(labels,location="northeastoutside")
hold off
end
```



Initialization Functions for Planners

Initialization function for `plannerHybridAStar`.

```
function planner = plannerHybridAStarWrapper(validator,minTurningRadius)
    map = validator.Map;
    ss = stateSpaceSE2;
    sv = validatorOccupancyMap(ss,Map=map);
    sv.ValidationDistance = validator.ValidationDistance;
    planner = plannerHybridAStar(sv,MinTurningRadius=minTurningRadius);
end
```

Initialization function for `plannerBiRRT`.

```
function planner = plannerBiRRTWrapper(sv)
    planner = plannerBiRRT(sv.StateSpace,sv);
    planner.EnableConnectHeuristic = true;
    planner.MaxIterations = 5e4;
    planner.MaxNumTreeNodees = 5e4;
    planner.MaxConnectionDistance = 5;
end
```

Initialization function for `plannerRRTStar`.

```
function planner = plannerRRTStarWrapper(sv)
    planner = plannerRRTStar(sv.StateSpace,sv);
    planner.MaxIterations = 5e5;
    planner.MaxNumTreeNodees = 5e5;
```

```
    planner.MaxConnectionDistance = 3.8;  
end
```

Initialization function for plannerRRT.

```
function planner = plannerRRTWrapper(sv)  
    planner = plannerRRT(sv.StateSpace,sv);  
    planner.MaxIterations = 5e5;  
    planner.MaxNumTreeNodees = 5e5;  
    planner.MaxConnectionDistance = 3.8;  
end
```

Generate Code for Path Planning Using Hybrid A Star

This example shows how to perform code generation to plan a collision-free path for a vehicle through a map using the Hybrid A* algorithm. After you verify the algorithm in MATLAB®, use the generated MEX file in the algorithm to visualize the planned path.

Write Algorithm to Plan Path

Create a function, `codegenPathPlanner`, that uses a `plannerHybridAStar` object to plan a path from the start pose to the goal pose in the map.

```
function path = codegenPathPlanner(mapData,startPose,goalPose)
% Copyright 2021 The MathWorks, Inc.
    %#codegen
    % Create a state space object
    stateSpace = stateSpaceSE2;

    % Create a binary occupancy map
    binMap = binaryOccupancyMap(mapData);

    % Construct a state validator object using the statespace and map object
    validator = validatorOccupancyMap(stateSpace,'Map',binMap);

    % Set the validation distance for the validator
    validator.ValidationDistance = 0.01;

    % Assign the state validator object to the plannerHybridAStar object
    planner = plannerHybridAStar(validator);

    % Compute a path for the given start and goal poses
    pathObj = plan(planner,startPose,goalPose);

    % Extract the path poses from the path object
    path = pathObj.States;
end
```

This function acts as a wrapper for a standard Hybrid A* path planning call. It accepts standard inputs and returns a collision-free path as an array. Because you cannot use a handle object as an input or output of a function that is supported for code generation, create the planner object inside the function. Save the `codegenPathPlanner` function in your current folder.

Verify Path Planning Algorithm in MATLAB

Verify the path planning algorithm in MATLAB before generating code.

Load a map into the workspace.

```
mapData = load("exampleMaps.mat").simpleMap;
```

Create a state space object.

```
stateSpace = stateSpaceSE2;
```

Create a binary occupancy map.

```
binMap = binaryOccupancyMap(mapData);
```

Construct a state validator object using the statespace and map object.

```
stateValidator = validatorOccupancyMap(stateSpace, 'Map', binMap);
```

Set the validation distance for the validator.

```
stateValidator.ValidationDistance = 0.01;
```

Initialize the `plannerHybridAStar` object with the state validator object.

```
planner = plannerHybridAStar(stateValidator);
```

Define start and goal poses as $[x \ y \ \theta]$ vectors. x and y specify the position in meters, and θ specifies the orientation angle in radians.

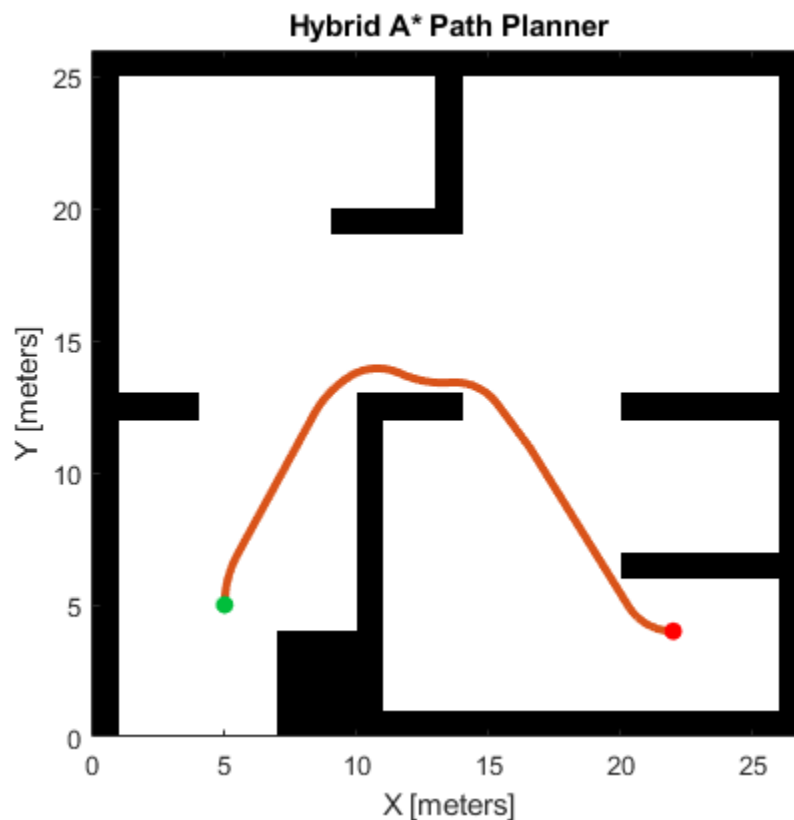
```
startPose = [5 5 pi/2];
goalPose = [22 4 0];
```

Plan a path from the start pose to the goal pose.

```
plan(planner, startPose, goalPose);
```

Visualize the path using the `show` function, and hide the expansion tree.

```
show(planner, Tree="off")
hold on
```



Generate Code for Path Planning Algorithm

You can use either the `codegen` (MATLAB Coder) function or the MATLAB Coder (MATLAB Coder) app to generate code. For this example, generate a MEX file by calling `codegen` at the MATLAB

command line. Specify sample input arguments for each input to the function using the `-args` option and `func_inputs` input argument.

Call the `codegen` function and specify the input arguments in a cell array. This function creates a separate `codegenPathPlanner_mex` function to use. You can also produce C code by using the `options` input argument. This step can take some time.

```
codegen codegenPathPlanner -args {mapData,startPose,goalPose}
```

Code generation successful.

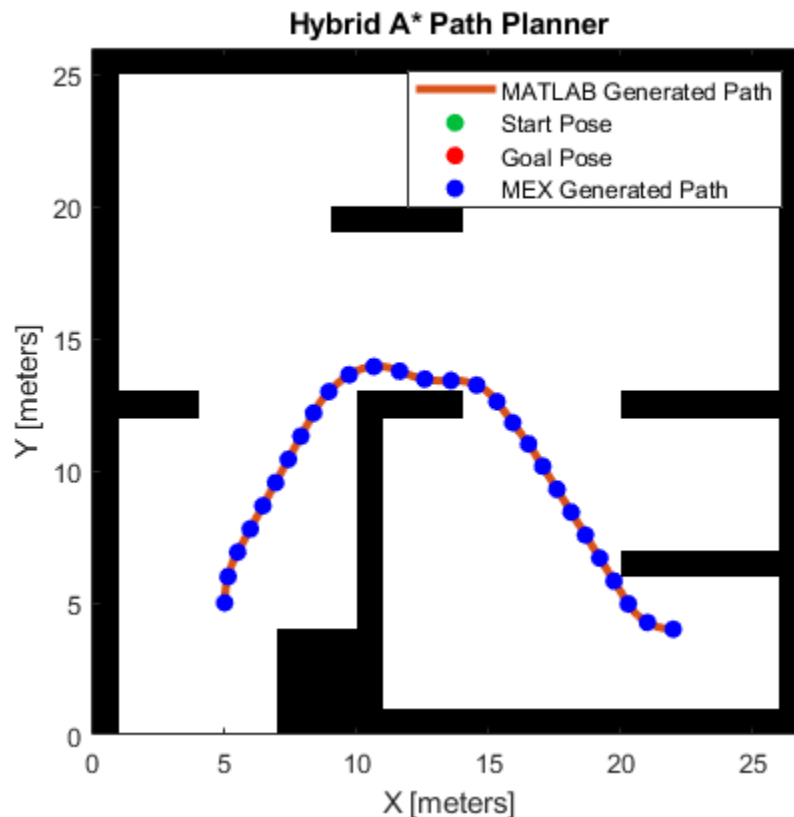
Verify Results Using Generated MEX Function

Plan the path by calling the MEX version of the path planning algorithm for the specified start pose, and goal pose, and map.

```
path = codegenPathPlanner_mex(mapData,startPose,goalPose);
```

Visualize the path computed by the MEX version of the path planning algorithm.

```
scatter(path(:,1),path(:,2),...
        Marker="o",...
        MarkerFaceColor="b",...
        MarkerEdgeColor="b")
legend("MATLAB Generated Path","Start Pose","Goal Pose","MEX Generated Path")
hold off
```



Check Performance of Generated Code

Compare the execution time of the generated MEX function to the execution time of your original function by using `timeit`.

```
time = timeit(@() codegenPathPlanner(mapData,startPose,goalPose))
time = 0.1287
mexTime = timeit(@() codegenPathPlanner_mex(mapData,startPose,goalPose))
mexTime = 0.0187
time/mexTime
ans = 6.8671
```

In this example, the MEX function runs more than five times faster. Results may vary for your system.

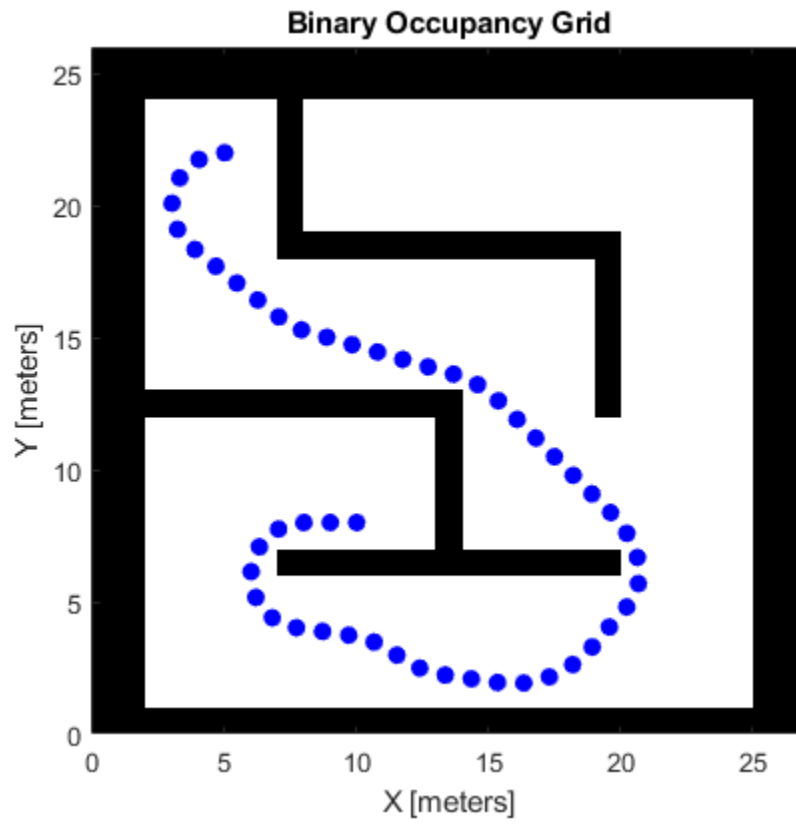
Plan Path in New Map Using Generated MEX Function

Plan a path for a new start and goal poses in a new map. The size of the new map must be the same as the map used to generate the MEX function.

```
mapNew = load("exampleMaze.mat").simpleMaze;
startPoseNew = [10 8 pi];
goalPoseNew = [5 22 0];
pathNew = codegenPathPlanner_mex(mapNew,startPoseNew,goalPoseNew);
```

Visualize the new path computed by the MEX function.

```
show(binaryOccupancyMap(mapNew))
hold on
scatter(pathNew(:,1),pathNew(:,2),...
        Marker="o",...
        MarkerFaceColor="b",...
        MarkerEdgeColor="b")
```



Estimate Position and Orientation of a Ground Vehicle

This example shows how to estimate the position and orientation of ground vehicles by fusing data from an inertial measurement unit (IMU) and a global positioning system (GPS) receiver.

Simulation Setup

Set the sampling rates. In a typical system, the accelerometer and gyroscope in the IMU run at relatively high sample rates. The complexity of processing data from those sensors in the fusion algorithm is relatively low. Conversely, the GPS runs at a relatively low sample rate and the complexity associated with processing it is high. In this fusion algorithm the GPS samples are processed at a low rate, and the accelerometer and gyroscope samples are processed together at the same high rate.

To simulate this configuration, the IMU (accelerometer and gyroscope) is sampled at 100 Hz, and the GPS is sampled at 10 Hz.

```
imuFs = 100;
gpsFs = 10;

% Define where on the Earth this simulation takes place using latitude,
% longitude, and altitude (LLA) coordinates.
localOrigin = [42.2825 -71.343 53.0352];

% Validate that the |gpsFs| divides |imuFs|. This allows the sensor sample
% rates to be simulated using a nested for loop without complex sample rate
% matching.

imuSamplesPerGPS = (imuFs/gpsFs);
assert(imuSamplesPerGPS == fix(imuSamplesPerGPS), ...
    'GPS sampling rate must be an integer factor of IMU sampling rate.');
```

Fusion Filter

Create the filter to fuse IMU + GPS measurements. The fusion filter uses an extended Kalman filter to track orientation (as a quaternion), position, velocity, and sensor biases.

The `insfilterNonholonomic` object has two main methods: `predict` and `fusegps`. The `predict` method takes the accelerometer and gyroscope samples from the IMU as input. Call the `predict` method each time the accelerometer and gyroscope are sampled. This method predicts the states forward one time step based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated in this step.

The `fusegps` method takes the GPS samples as input. This method updates the filter states based on the GPS sample by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated in this step, this time using the Kalman gain as well.

The `insfilterNonholonomic` object has two main properties: `IMUSampleRate` and `DecimationFactor`. The ground vehicle has two velocity constraints that assume it does not bounce off the ground or slide on the ground. These constraints are applied using the extended Kalman filter update equations. These updates are applied to the filter states at a rate of `IMUSampleRate/DecimationFactor` Hz.

```
gndFusion = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
    'IMUSampleRate', imuFs, ...
```

```
    'ReferenceLocation', localOrigin, ...  
    'DecimationFactor', 2);
```

Create Ground Vehicle Trajectory

The `waypointTrajectory` object calculates pose based on specified sampling rate, waypoints, times of arrival, and orientation. Specify the parameters of a circular trajectory for the ground vehicle.

```
% Trajectory parameters  
r = 8.42; % (m)  
speed = 2.50; % (m/s)  
center = [0, 0]; % (m)  
initialYaw = 90; % (degrees)  
numRevs = 2;  
  
% Define angles theta and corresponding times of arrival t.  
revTime = 2*pi*r / speed;  
theta = (0:pi/2:2*pi*numRevs).';  
t = linspace(0, revTime*numRevs, numel(theta)).';  
  
% Define position.  
x = r .* cos(theta) + center(1);  
y = r .* sin(theta) + center(2);  
z = zeros(size(x));  
position = [x, y, z];  
  
% Define orientation.  
yaw = theta + deg2rad(initialYaw);  
yaw = mod(yaw, 2*pi);  
pitch = zeros(size(yaw));  
roll = zeros(size(yaw));  
orientation = quaternion([yaw, pitch, roll], 'euler', ...  
    'ZYX', 'frame');  
  
% Generate trajectory.  
groundTruth = waypointTrajectory('SampleRate', imuFs, ...  
    'Waypoints', position, ...  
    'TimeOfArrival', t, ...  
    'Orientation', orientation);  
  
% Initialize the random number generator used to simulate sensor noise.  
rng('default');
```

GPS Receiver

Set up the GPS at the specified sample rate and reference location. The other parameters control the nature of the noise in the output signal.

```
gps = gpsSensor('UpdateRate', gpsFs, 'ReferenceFrame', 'ENU');  
gps.ReferenceLocation = localOrigin;  
gps.DecayFactor = 0.5; % Random walk noise parameter  
gps.HorizontalPositionAccuracy = 1.0;  
gps.VerticalPositionAccuracy = 1.0;  
gps.VelocityAccuracy = 0.1;
```

IMU Sensors

Typically, ground vehicles use a 6-axis IMU sensor for pose estimation. To model an IMU sensor, define an IMU sensor model containing an accelerometer and gyroscope. In a real-world application,

the two sensors could come from a single integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors.

```
imu = imuSensor('accel-gyro', ...
    'ReferenceFrame', 'ENU', 'SampleRate', imuFs);

% Accelerometer
imu.Accelerometer.MeasurementRange = 19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
imu.Gyroscope.NoiseDensity = deg2rad(0.025);
```

Initialize the States of the `insfilterNonholonomic`

The states are:

States	Units	Index
Orientation (quaternion parts)		1:4
Gyroscope Bias (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s ²	14:16

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
% Get the initial ground truth pose from the first sample of the trajectory
% and release the ground truth trajectory to ensure the first sample is not
% skipped during simulation.
[initialPos, initialAtt, initialVel] = groundTruth();
reset(groundTruth);

% Initialize the states of the filter
gndFusion.State(1:4) = compact(initialAtt).';
gndFusion.State(5:7) = imu.Gyroscope.ConstantBias;
gndFusion.State(8:10) = initialPos.';
gndFusion.State(11:13) = initialVel.';
gndFusion.State(14:16) = imu.Accelerometer.ConstantBias;
```

Initialize the Variances of the `insfilterNonholonomic`

The measurement noises describe how much noise is corrupting the GPS reading based on the `gpsSensor` parameters and how much uncertainty is in the vehicle dynamic model.

The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter.

```
% Measurement noises
Rvel = gps.VelocityAccuracy.^2;
Rpos = gps.HorizontalPositionAccuracy.^2;

% The dynamic model of the ground vehicle for this filter assumes there is
% no side slip or skid during movement. This means that the velocity is
% constrained to only the forward body axis. The other two velocity axis
```

```
% readings are corrected with a zero measurement weighted by the
% |ZeroVelocityConstraintNoise| parameter.
gndFusion.ZeroVelocityConstraintNoise = 1e-2;

% Process noises
gndFusion.GyroscopeNoise = 4e-6;
gndFusion.GyroscopeBiasNoise = 4e-14;
gndFusion.AccelerometerNoise = 4.8e-2;
gndFusion.AccelerometerBiasNoise = 4e-14;

% Initial error covariance
gndFusion.StateCovariance = 1e-9*ones(16);
```

Initialize Scopes

The `HelperScrollingPlotter` scope enables plotting of variables over time. It is used here to track errors in pose. The `HelperPoseViewer` scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to `false`.

```
useErrScope = true; % Turn on the streaming error plot
usePoseView = true; % Turn on the 3D pose viewer
```

```
if useErrScope
    errscope = HelperScrollingPlotter( ...
        'NumInputs', 4, ...
        'TimeSpan', 10, ...
        'SampleRate', imuFs, ...
        'YLabel', {'degrees', ...
        'meters', ...
        'meters', ...
        'meters'}}, ...
        'Title', {'Quaternion Distance', ...
        'Position X Error', ...
        'Position Y Error', ...
        'Position Z Error'}}, ...
        'YLimits', ...
        [-1, 1
        -1, 1
        -1, 1
        -1, 1]);
end
```

```
if usePoseView
    viewer = HelperPoseViewer( ...
        'XPositionLimits', [-15, 15], ...
        'YPositionLimits', [-15, 15], ...
        'ZPositionLimits', [-5, 5], ...
        'ReferenceFrame', 'ENU');
end
```

Simulation Loop

The main simulation loop is a while loop with a nested for loop. The while loop executes at the `gpsFs`, which is the GPS measurement rate. The nested for loop executes at the `imuFs`, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

```

totalSimTime = 30; % seconds

% Log data for final metric computation.
numsamples = floor(min(t(end), totalSimTime) * gpsFs);
truePosition = zeros(numsamples,3);
trueOrientation = quaternion.zeros(numsamples,1);
estPosition = zeros(numsamples,3);
estOrientation = quaternion.zeros(numsamples,1);

idx = 0;

for sampleIdx = 1:numsamples
    % Predict loop at IMU update frequency.
    for i = 1:imuSamplesPerGPS
        if ~isDone(groundTruth)
            idx = idx + 1;

            % Simulate the IMU data from the current pose.
            [truePosition(idx,:), trueOrientation(idx,:), ...
             trueVel, trueAcc, trueAngVel] = groundTruth();
            [accelData, gyroData] = imu(trueAcc, trueAngVel, ...
                                         trueOrientation(idx,:));

            % Use the predict method to estimate the filter state based
            % on the accelData and gyroData arrays.
            predict(gndFusion, accelData, gyroData);

            % Log the estimated orientation and position.
            [estPosition(idx,:), estOrientation(idx,:)] = pose(gndFusion);

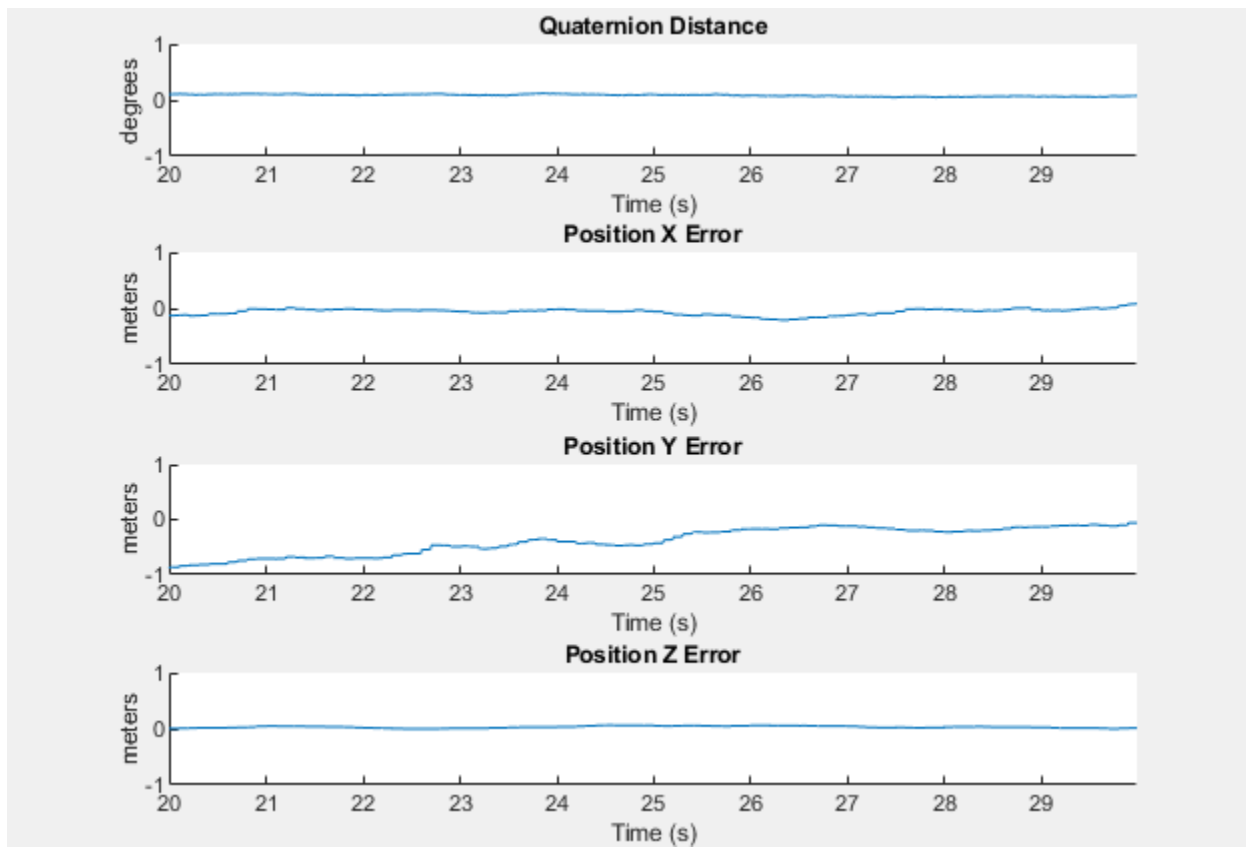
            % Compute the errors and plot.
            if useErrScope
                orientErr = rad2deg( ...
                    dist(estOrientation(idx,:), trueOrientation(idx,:)));
                posErr = estPosition(idx,:) - truePosition(idx,:);
                errsScope(orientErr, posErr(1), posErr(2), posErr(3));
            end

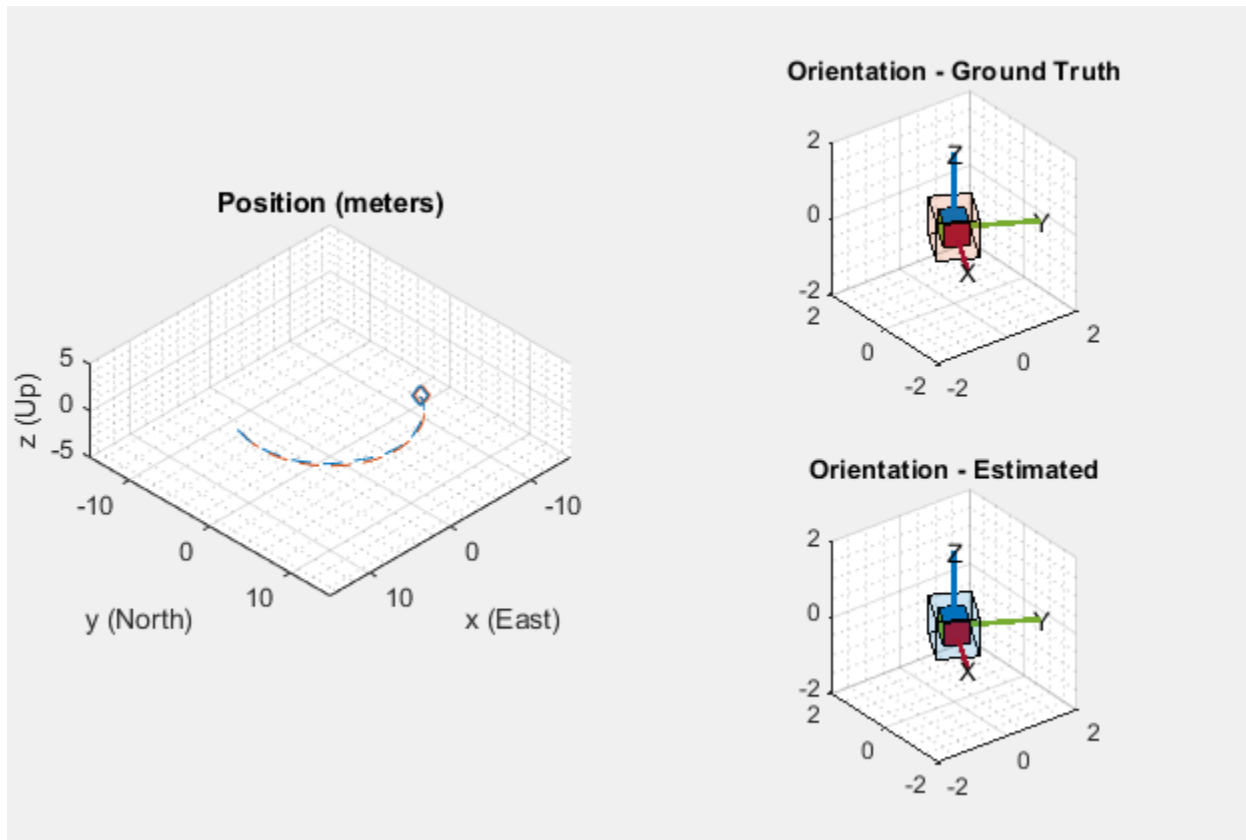
            % Update the pose viewer.
            if usePoseView
                viewer(estPosition(idx,:), estOrientation(idx,:), ...
                    truePosition(idx,:), estOrientation(idx,:));
            end
        end
    end
end

if ~isDone(groundTruth)
    % This next step happens at the GPS sample rate.
    % Simulate the GPS output based on the current pose.
    [lla, gpsVel] = gps(truePosition(idx,:), trueVel);

    % Update the filter states based on the GPS data.
    fusegps(gndFusion, lla, Rpos, gpsVel, Rvel);
end
end

```





Error Metric Computation

Position and orientation were logged throughout the simulation. Now compute an end-to-end root mean squared error for both position and orientation.

```
posd = estPosition - truePosition;

% For orientation, quaternion distance is a much better alternative to
% subtracting Euler angles, which have discontinuities. The quaternion
% distance can be computed with the |dist| function, which gives the
% angular difference in orientation in radians. Convert to degrees for
% display in the command window.

quadd = rad2deg(dist(estOrientation, trueOrientation));

% Display RMS errors in the command window.
fprintf('\n\nEnd-to-End Simulation Position RMS Error\n');

End-to-End Simulation Position RMS Error

msepos = sqrt(mean(posd.^2));
fprintf('\tX: %.2f , Y: %.2f, Z: %.2f (meters)\n\n', msepos(1), ...
    msepos(2), msepos(3));

    X: 1.16 , Y: 0.99, Z: 0.03 (meters)

fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
```

End-to-End Quaternion Distance RMS Error (degrees)

```
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));  
0.09 (degrees)
```

Pose Estimation From Asynchronous Sensors

This example shows how you might fuse sensors at different rates to estimate pose. Accelerometer, gyroscope, magnetometer and GPS are used to determine orientation and position of a vehicle moving along a circular path. You can use controls on the figure window to vary sensor rates and experiment with sensor dropout while seeing the effect on the estimated pose.

Simulation Setup

Load prerecorded sensor data. The sensor data is based on a circular trajectory created using the `waypointTrajectory` class. The sensor values were created using the `gpsSensor` and `imuSensor` classes. The `CircularTrajectorySensorData.mat` file used here can be generated with the `generateCircularTrajSensorData` function.

```
ld = load('CircularTrajectorySensorData.mat');

Fs = ld.Fs; % maximum MARG rate
gpsFs = ld.gpsFs; % maximum GPS rate
ratio = Fs./gpsFs;
refloc = ld.refloc;

trajOrient = ld.trajData.Orientation;
trajVel = ld.trajData.Velocity;
trajPos = ld.trajData.Position;
trajAcc = ld.trajData.Acceleration;
trajAngVel = ld.trajData.AngularVelocity;

accel = ld.accel;
gyro = ld.gyro;
mag = ld.mag;
lla = ld.lla;
gpsvel = ld.gpsvel;
```

Fusion Filter

Create an `insfilterAsync` to fuse IMU + GPS measurements. This fusion filter uses a continuous-discrete extended Kalman filter (EKF) to track orientation (as a quaternion), angular velocity, position, velocity, acceleration, sensor biases, and the geomagnetic vector.

This `insfilterAsync` has several methods to process sensor data: `fuseaccel`, `fusegyro`, `fusemag` and `fusegps`. Because `insfilterAsync` uses a continuous-discrete EKF, the `predict` method can step the filter forward an arbitrary amount of time.

```
fusionfilt = insfilterAsync('ReferenceLocation', refloc);
```

Initialize the State Vector of the `insfilterAsync`

The `insfilterAsync` tracks the pose states in a 28-element vector. The states are:

States	Units	Index
Orientation (quaternion parts)		1:4
Angular Velocity (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Acceleration (NED)	m/s ²	14:16
Accelerometer Bias (XYZ)	m/s ²	17:19
Gyroscope Bias (XYZ)	rad/s	20:22

```
Geomagnetic Field Vector (NED)    uT    23:25
Magnetometer Bias (XYZ)         uT    26:28
```

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
Nav = 100;
initstate = zeros(28,1);
initstate(1:4) = compact( meanrot(trajOrient(1:Nav)));
initstate(5:7) = mean( trajAngVel(10:Nav,:), 1);
initstate(8:10) = mean( trajPos(1:Nav,:), 1);
initstate(11:13) = mean( trajVel(1:Nav,:), 1);
initstate(14:16) = mean( trajAcc(1:Nav,:), 1);
initstate(23:25) = ld.magField;

% The gyroscope bias initial value estimate is low for the Z-axis. This is
% done to illustrate the effects of fusing the magnetometer in the
% simulation.
initstate(20:22) = deg2rad([3.125 3.125 3.125]);
fusionfilt.State = initstate;
```

Set the Process Noise Values of the `insfilterAsync`

The process noise variance describes the uncertainty of the motion model the filter uses.

```
fusionfilt.QuaternionNoise = 1e-2;
fusionfilt.AngularVelocityNoise = 100;
fusionfilt.AccelerationNoise = 100;
fusionfilt.MagnetometerBiasNoise = 1e-7;
fusionfilt.AccelerometerBiasNoise = 1e-7;
fusionfilt.GyroscopeBiasNoise = 1e-7;
```

Define the Measurement Noise Values Used to Fuse Sensor Data

Each sensor has some noise in the measurements. These values can typically be found on a sensor's datasheet.

```
Rmag = 0.4;
Rvel = 0.01;
Racc = 610;
Rgyro = 0.76e-5;
Rpos = 3.4;
```

```
fusionfilt.StateCovariance = diag(1e-3*ones(28,1));
```

Initialize Scopes

The `HelperScrollingPlotter` scope enables plotting of variables over time. It is used here to track errors in pose. The `PoseViewerWithSwitches` scope allows 3D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to false.

```
useErrScope = true; % Turn on the streaming error plot.
usePoseView = true; % Turn on the 3D pose viewer.
if usePoseView
    posescope = PoseViewerWithSwitches(...
        'XPositionLimits', [-30 30], ...
        'YPositionLimits', [-30, 30], ...
        'ZPositionLimits', [-10 10]);
```

```

end
f = gcf;

if useErrScope
    errscope = HelperScrollingPlotter(...
        'NumInputs', 4, ...
        'TimeSpan', 10, ...
        'SampleRate', Fs, ...
        'YLabel', {'degrees', ...
        'meters', ...
        'meters', ...
        'meters'}, ...
        'Title', {'Quaternion Distance', ...
        'Position X Error', ...
        'Position Y Error', ...
        'Position Z Error'}, ...
        'YLimits', ...
        [-1, 30
        -2, 2
        -2 2
        -2 2]);
end

```

Simulation Loop

The simulation of the fusion algorithm allows you to inspect the effects of varying sensor sample rates. Further, fusion of individual sensors can be prevented by unchecking the corresponding checkbox. This can be used to simulate sensor dropout.

Some configurations produce dramatic results. For example, turning off the GPS sensor causes the position estimate to drift quickly. Turning off the magnetometer sensor will cause the orientation estimate to slowly deviate from the ground truth as the estimate rotates too fast. Conversely, if the gyroscope is turned off and the magnetometer is turned on, the estimated orientation shows a wobble and lacks the smoothness present if both sensors are used.

Turning all sensors on but setting them to run at the lowest rate produces an estimate that visibly deviates from the ground truth and then snaps back to a more correct result when sensors are fused. This is most easily seen in the `HelperScrollingPlotter` of the running estimate errors.

The main simulation runs at 100 Hz. Each iteration inspects the checkboxes on the figure window and, if the sensor is enabled, fuses the data for that sensor at the appropriate rate.

```

for ii=1:size(accel,1)
    fusionfilt.predict(1./Fs);

    % Fuse Accelerometer
    if (f.UserData.Accelerometer) && ...
        mod(ii, fix(Fs/f.UserData.AccelerometerSampleRate)) == 0

        fusionfilt.fuseaccel(accel(ii,:), Raccel);
    end

    % Fuse Gyroscope
    if (f.UserData.Gyroscope) && ...
        mod(ii, fix(Fs/f.UserData.GyroscopeSampleRate)) == 0

        fusionfilt.fusegyro(gyro(ii,:), Rgyro);
    end
end

```

```

end

% Fuse Magnetometer
if (f.UserData.Magnetometer) && ...
    mod(ii, fix(Fs/f.UserData.MagnetometerSampleRate)) == 0

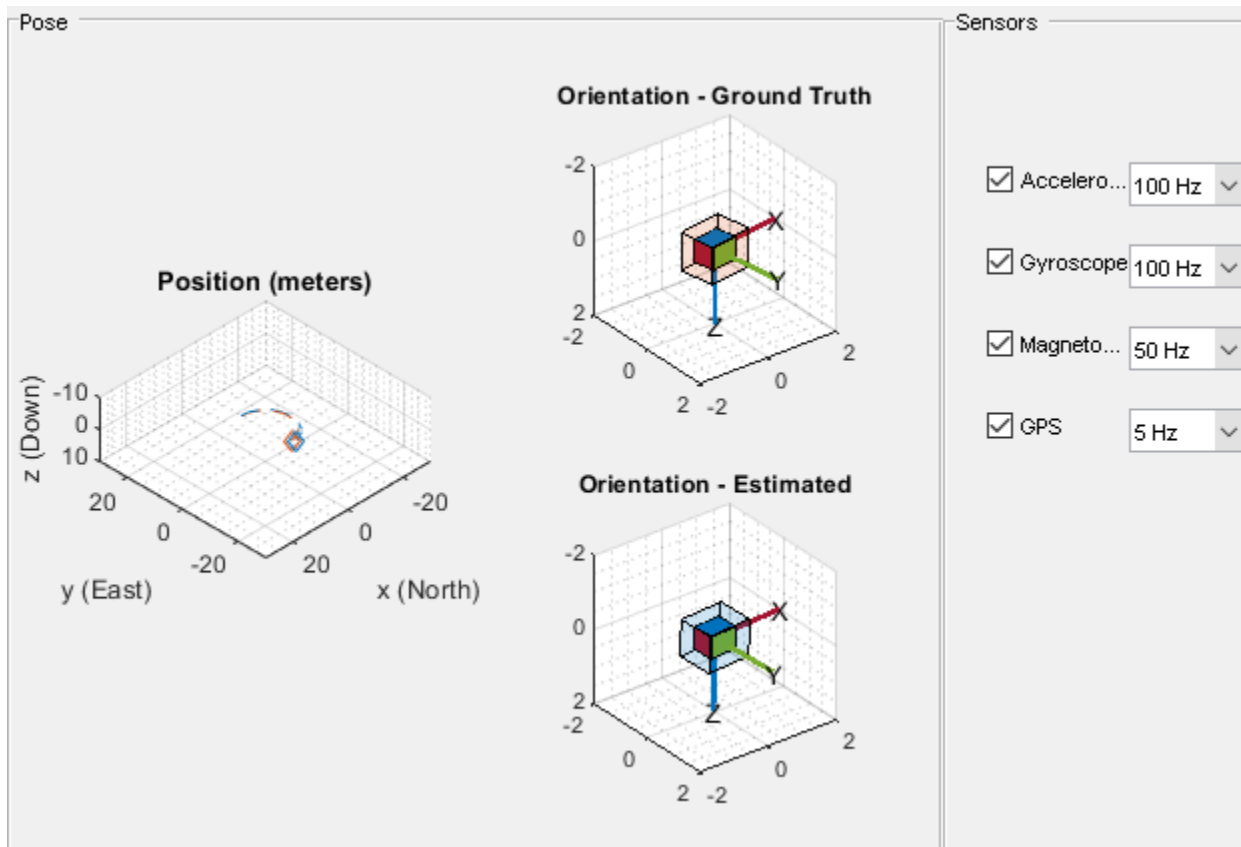
    fusionfilt.fusemag(mag(ii,:), Rmag);
end

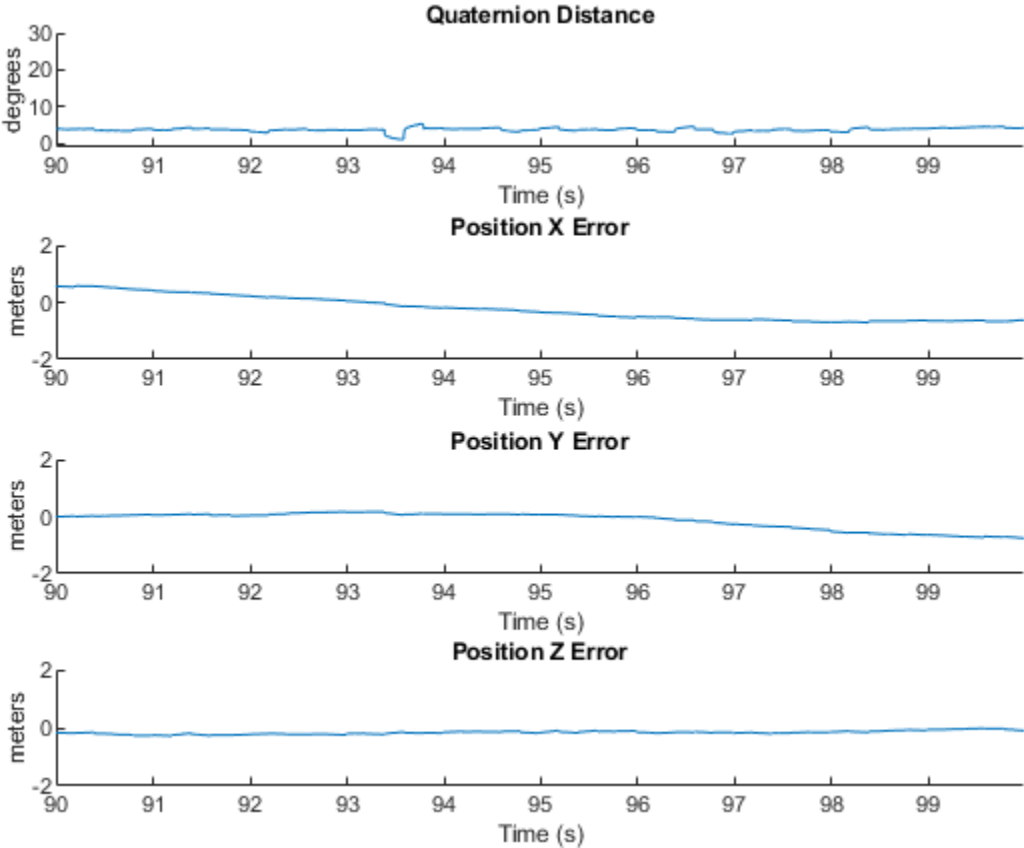
% Fuse GPS
if (f.UserData.GPS) && mod(ii, fix(Fs/f.UserData.GPSSampleRate)) == 0
    fusionfilt.fusegps(lla(ii,:), Rpos, gpsvel(ii,:), Rvel);
end

% Plot the pose error
[p,q] = pose(fusionfilt);
posescope(p, q, trajPos(ii,:), trajOrient(ii));

orientErr = rad2deg(dist(q, trajOrient(ii) ));
posErr = p - trajPos(ii,:);
errscope(orientErr, posErr(1), posErr(2), posErr(3));
end

```





Conclusion

The `insfilterAsync` allows for various and varying sample rates. The quality of the estimated outputs depends heavily on individual sensor fusion rates. Any sensor dropout will have a profound effect on the output.

Inertial Sensor Noise Analysis Using Allan Variance

This example shows how to use the Allan variance to determine noise parameters of a MEMS gyroscope. These parameters can be used to model the gyroscope in simulation. The gyroscope measurement is modeled as:

$$\Omega(t) = \Omega_{Ideal}(t) + Bias_N(t) + Bias_B(t) + Bias_K(t)$$

The three noise parameters N (angle random walk), K (rate random walk), and B (bias instability) are estimated using data logged from a stationary gyroscope.

Background

Allan variance was originally developed by David W. Allan to measure the frequency stability of precision oscillators. It can also be used to identify various noise sources present in stationary gyroscope measurements. Consider L samples of data from a gyroscope with a sample time of τ_0 . Form data clusters of durations $\tau_0, 2\tau_0, \dots, m\tau_0$, ($m < (L - 1)/2$) and obtain the averages of the sum of the data points contained in each cluster over the length of the cluster. The Allan variance is defined as the two-sample variance of the data cluster averages as a function of cluster time. This example uses the overlapping Allan variance estimator. This means that the calculated clusters are overlapping. The estimator performs better than non-overlapping estimators for larger values of L .

Allan Variance Calculation

The Allan variance is calculated as follows:

Log L stationary gyroscope samples with a sample period τ_0 . Let Ω be the logged samples.

```
% Load logged data from one axis of a three-axis gyroscope. This recording
% was done over a six hour period with a 100 Hz sampling rate.
load('LoggedSingleAxisGyroscope', 'omega', 'Fs')
t0 = 1/Fs;
```

For each sample, calculate the output angle θ :

$$\theta(t) = \int^t \Omega(t') dt'$$

For discrete samples, the cumulative sum multiplied by τ_0 can be used.

```
theta = cumsum(omega, 1)*t0;
```

Next, calculate the Allan variance:

$$\sigma^2(\tau) = \frac{1}{2\tau^2} \langle (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2 \rangle$$

where $\tau = m\tau_0$ and $\langle \rangle$ is the ensemble average.

The ensemble average can be expanded to:

$$\sigma^2(\tau) = \frac{1}{2\tau^2(L - 2m)} \sum_{k=1}^{L-2m} (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2$$


```

maxNumM = 100;
L = size(theta, 1);
maxM = 2.^floor(log2(L/2));
m = logspace(log10(1), log10(maxM), maxNumM).';
m = ceil(m); % m must be an integer.
m = unique(m); % Remove duplicates.

tau = m*t0;

avar = zeros(numel(m), 1);
for i = 1:numel(m)
    mi = m(i);
    avar(i,:) = sum( ...
        (theta(1+2*mi:L) - 2*theta(1+mi:L-mi) + theta(1:L-2*mi)).^2, 1);
end
avar = avar ./ (2*tau.^2 .* (L - 2*m));

```

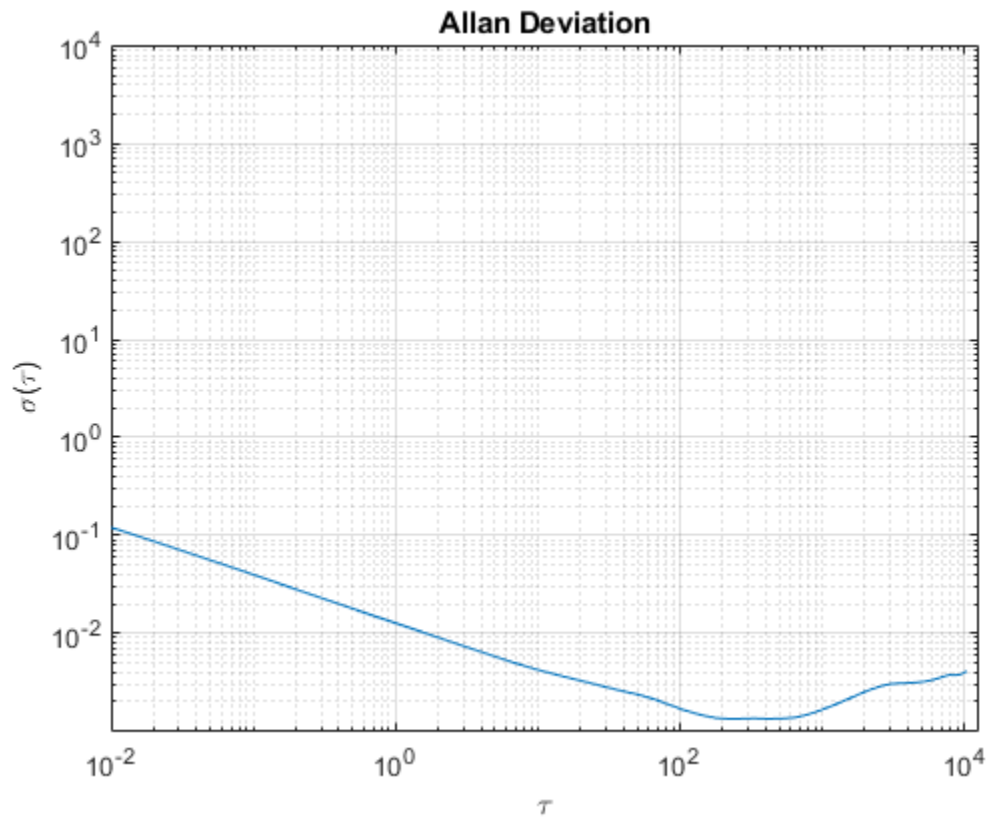
Finally, the Allan deviation $\sigma(t) = \sqrt{\sigma^2(t)}$ is used to determine the gyroscope noise parameters.

```

adev = sqrt(avar);

figure
loglog(tau, adev)
title('Allan Deviation')
xlabel('\tau');
ylabel('\sigma(\tau)')
grid on
axis equal

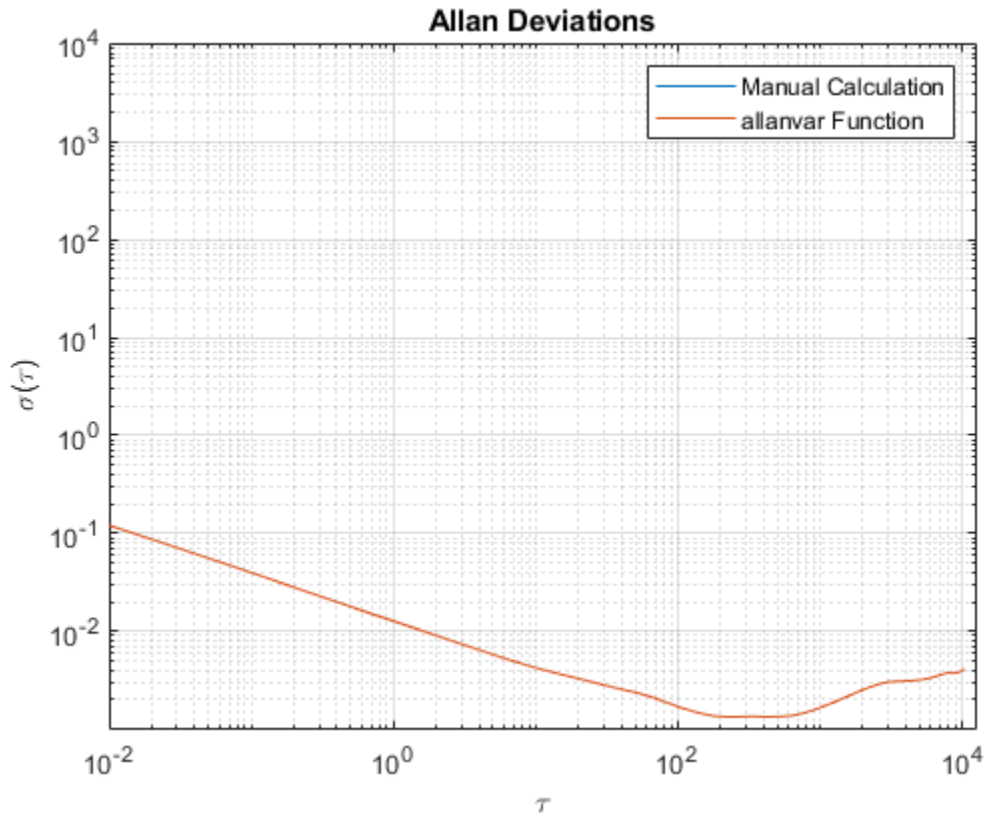
```



The Allan variance can also be calculated using the `allanvar` function.

```
[avarFromFunc, tauFromFunc] = allanvar(omega, m, Fs);  
adevFromFunc = sqrt(avarFromFunc);
```

```
figure  
loglog(tau, adev, tauFromFunc, adevFromFunc);  
title('Allan Deviations')  
xlabel('\tau')  
ylabel('\sigma(\tau)')  
legend('Manual Calculation', 'allanvar Function')  
grid on  
axis equal
```



Noise Parameter Identification

To obtain the noise parameters for the gyroscope, use the following relationship between the Allan variance and the two-sided power spectral density (PSD) of the noise parameters in the original data set Ω . The relationship is:

$$\sigma^2(\tau) = 4 \int_0^{\infty} S_{\Omega}(f) \frac{\sin^4(\pi f \tau)}{(\pi f \tau)^2} df$$

From the above equation, the Allan variance is proportional to the total noise power of the gyroscope when passed through a filter with a transfer function of $\sin^4(x)/(x)^2$. This transfer function arises from the operations done to create and operate on the clusters.

Using this transfer function interpretation, the filter bandpass depends on τ . This means that different noise parameters can be identified by changing the filter bandpass, or varying τ .

Angle Random Walk

The angle random walk is characterized by the white noise spectrum of the gyroscope output. The PSD is represented by:

$$S_{\Omega}(f) = N^2$$

where

N = angle random walk coefficient

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{N^2}{\tau}$$

The above equation is a line with a slope of -1/2 when plotted on a log-log plot of $\sigma(\tau)$ versus τ . The value of N can be read directly off of this line at $\tau = 1$. The units of N are $(rad/s)/\sqrt{Hz}$.

```
% Find the index where the slope of the log-scaled Allan deviation is equal  
% to the slope specified.
```

```
slope = -0.5;  
logtau = log10(tau);  
logadev = log10(adev);  
dlogadev = diff(logadev) ./ diff(logtau);  
[~, i] = min(abs(dlogadev - slope));
```

```
% Find the y-intercept of the line.
```

```
b = logadev(i) - slope*logtau(i);
```

```
% Determine the angle random walk coefficient from the line.
```

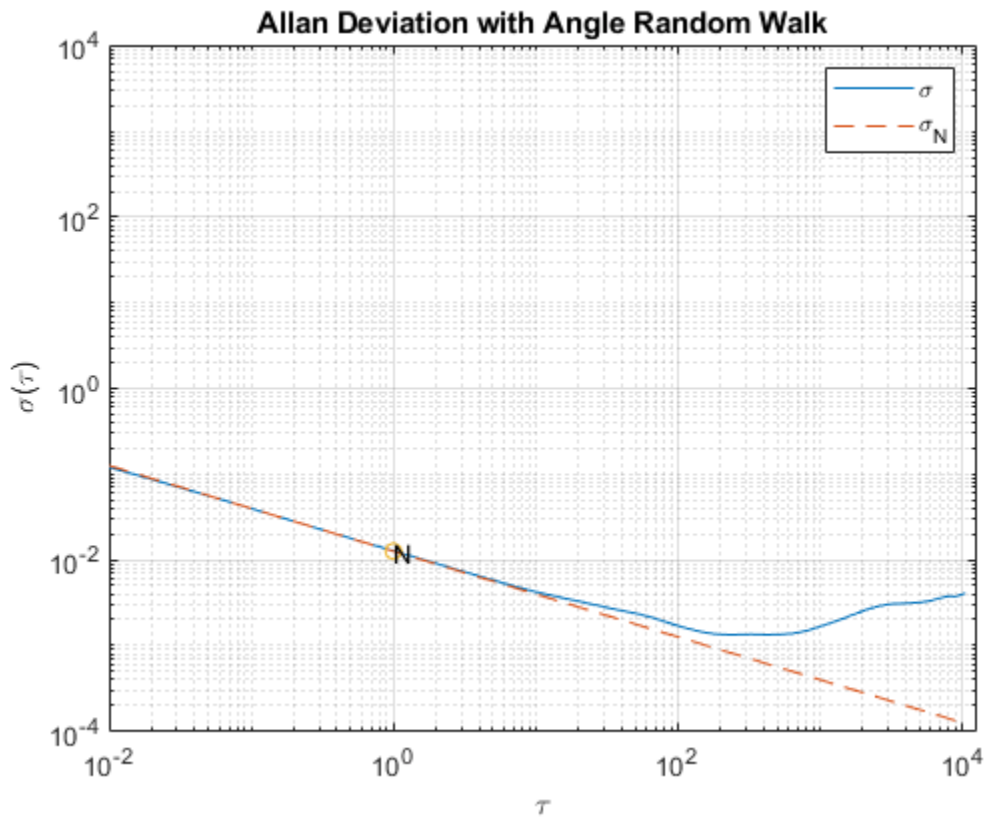
```
logN = slope*log(1) + b;  
N = 10^logN
```

```
% Plot the results.
```

```
tauN = 1;  
lineN = N ./ sqrt(tau);  
figure  
loglog(tau, adev, tau, lineN, '--', tauN, N, 'o')  
title('Allan Deviation with Angle Random Walk')  
xlabel('\tau')  
ylabel('\sigma(\tau)')  
legend('\sigma', '\sigma_N')  
text(tauN, N, 'N')  
grid on  
axis equal
```

```
N =
```

```
0.0126
```



Rate Random Walk

The rate random walk is characterized by the red noise (Brownian noise) spectrum of the gyroscope output. The PSD is represented by:

$$S_{\Omega}(f) = \left(\frac{K}{2\pi}\right)^2 \frac{1}{f^2}$$

where

K = rate random walk coefficient

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{K^2\tau}{3}$$

The above equation is a line with a slope of 1/2 when plotted on a log-log plot of $\sigma(\tau)$ versus τ . The value of K can be read directly off of this line at $\tau = 3$. The units of K are $(\text{rad/s})\sqrt{\text{Hz}}$.

`% Find the index where the slope of the log-scaled Allan deviation is equal
% to the slope specified.`

`slope = 0.5;`

`logtau = log10(tau);`

`logadev = log10(adev);`

```

dlogadev = diff(logadev) ./ diff(logtau);
[~, i] = min(abs(dlogadev - slope));

% Find the y-intercept of the line.
b = logadev(i) - slope*logtau(i);

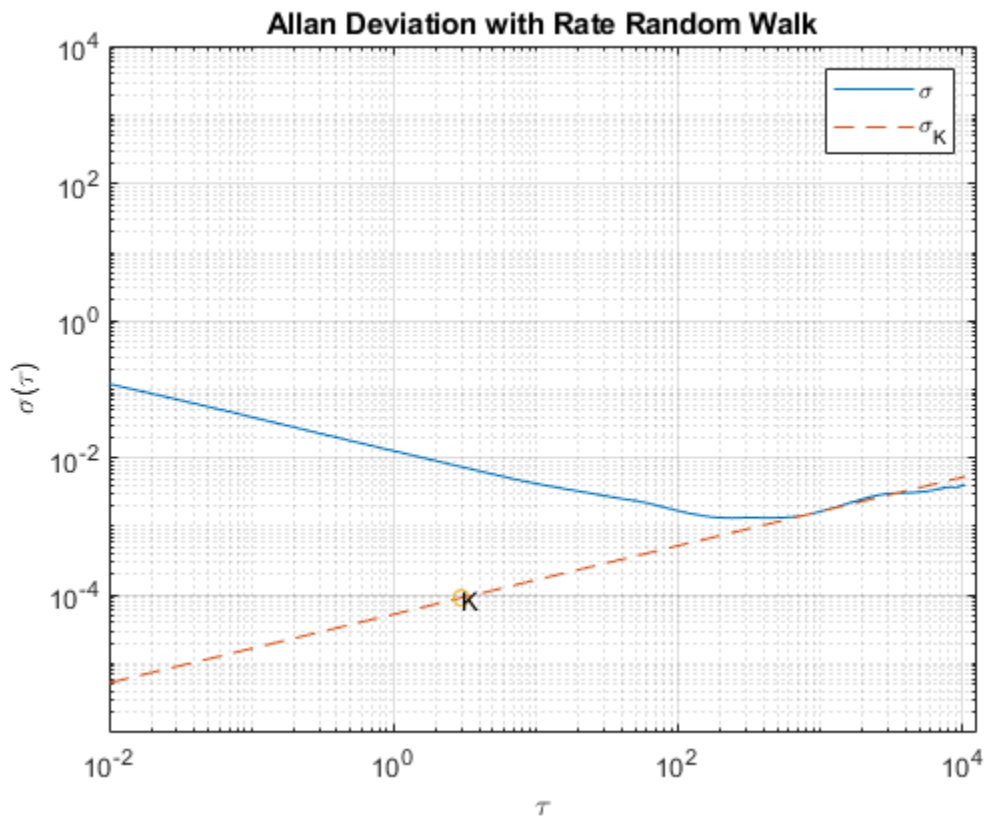
% Determine the rate random walk coefficient from the line.
logK = slope*log10(3) + b;
K = 10^logK

% Plot the results.
tauK = 3;
lineK = K .* sqrt(tau/3);
figure
loglog(tau, adev, tau, lineK, '--', tauK, K, 'o')
title('Allan Deviation with Rate Random Walk')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma', '\sigma_K')
text(tauK, K, 'K')
grid on
axis equal

```

K =

9.0679e-05



Bias Instability

The bias instability is characterized by the pink noise (flicker noise) spectrum of the gyroscope output. The PSD is represented by:

$$S_{\Omega}(f) = \begin{cases} \left(\frac{B^2}{2\pi}\right)^{\frac{1}{f}} & : f \leq f_0 \\ 0 & : f > f_0 \end{cases}$$

where

B = bias instability coefficient

f_0 = cut-off frequency

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{2B^2}{\pi} \left[\ln 2 + -\frac{\sin^3 x}{2x^2} (\sin x + 4x \cos x) + Ci(2x) - Ci(4x) \right]$$

where

$$x = \pi f_0 \tau$$

Ci = cosine-integral function

When τ is much longer than the inverse of the cutoff frequency, the PSD equation is:

$$\sigma^2(\tau) = \frac{2B^2}{\pi} \ln 2$$

The above equation is a line with a slope of 0 when plotted on a log-log plot of $\sigma(\tau)$ versus τ . The value of B can be read directly off of this line with a scaling of $\sqrt{\frac{2 \ln 2}{\pi}} \approx 0.664$. The units of B are *rad/s*.

% Find the index where the slope of the log-scaled Allan deviation is equal to the slope specified.

```
slope = 0;
logtau = log10(tau);
logadev = log10(adev);
dlogadev = diff(logadev) ./ diff(logtau);
[~, i] = min(abs(dlogadev - slope));
```

% Find the y-intercept of the line.

```
b = logadev(i) - slope*logtau(i);
```

% Determine the bias instability coefficient from the line.

```
scfB = sqrt(2*log(2)/pi);
logB = b - log10(scfB);
B = 10^logB
```

% Plot the results.

```
tauB = tau(i);
lineB = B * scfB * ones(size(tau));
```

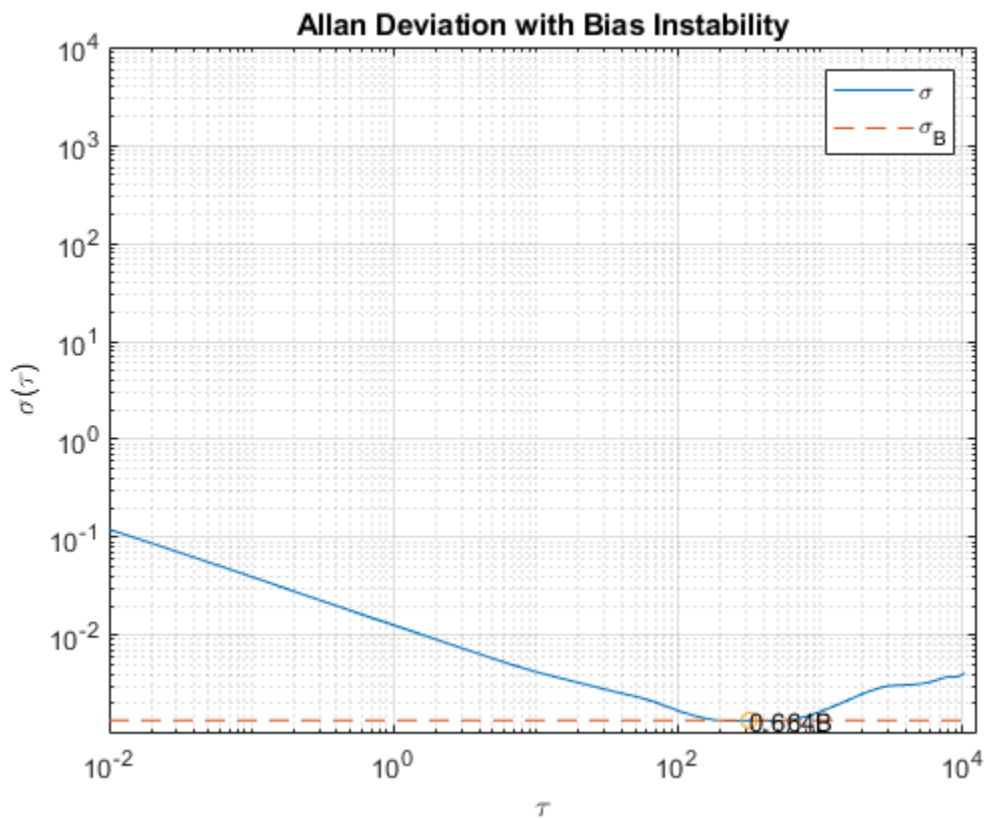
```

figure
loglog(tau, adev, tau, lineB, '--', tauB, scfB*B, 'o')
title('Allan Deviation with Bias Instability')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma', '\sigma_B')
text(tauB, scfB*B, '0.664B')
grid on
axis equal

```

B =

0.0020



Now that all the noise parameters have been calculated, plot the Allan deviation with all of the lines used for quantifying the parameters.

```

tauParams = [tauN, tauK, tauB];
params = [N, K, scfB*B];
figure
loglog(tau, adev, tau, [lineN, lineK, lineB], '--', ...
      tauParams, params, 'o')
title('Allan Deviation with Noise Parameters')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma (rad/s)', '\sigma_N ((rad/s)/\sqrt{Hz})$', ...

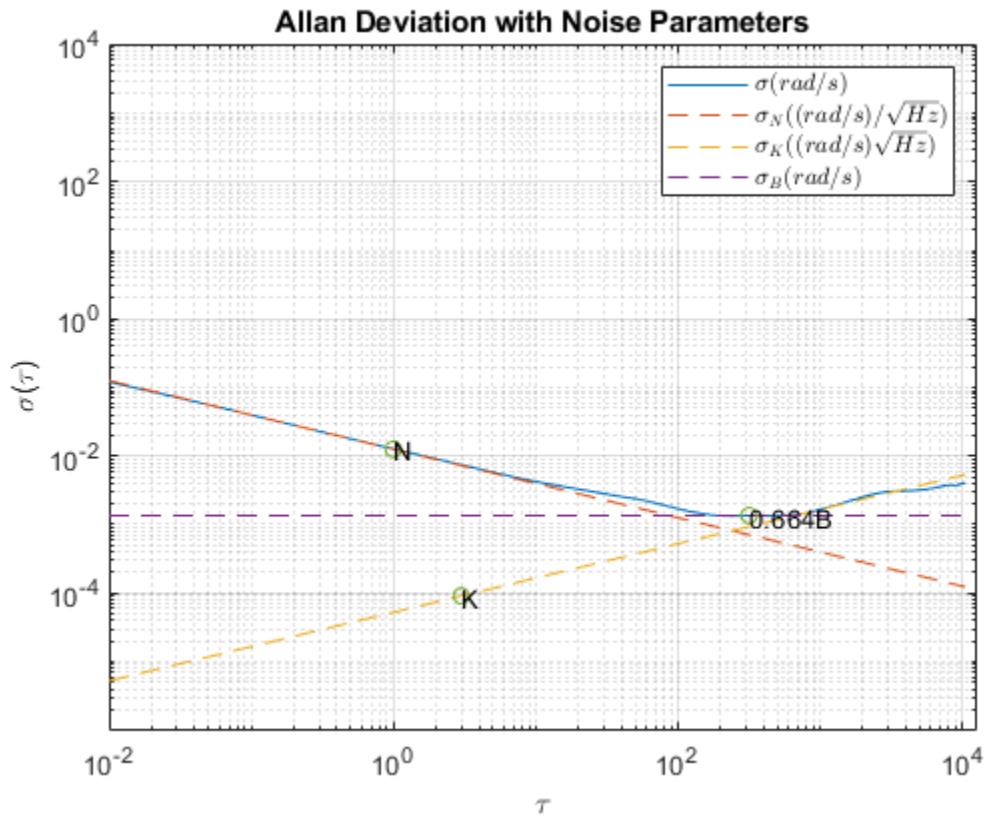
```



```

'$\sigma_K ((rad/s)\sqrt{Hz})$', '$\sigma_B (rad/s)$', 'Interpreter', 'latex')
text(tauParams, params, {'N', 'K', '0.664B'})
grid on
axis equal

```



Gyroscope Simulation

Use the `imuSensor` object to simulate gyroscope measurements with the noise parameters identified above.

```

% Simulating the gyroscope measurements takes some time. To avoid this, the
% measurements were generated and saved to a MAT-file. By default, this
% example uses the MAT-file. To generate the measurements instead, change
% this logical variable to true.
generateSimulatedData = false;

```

```

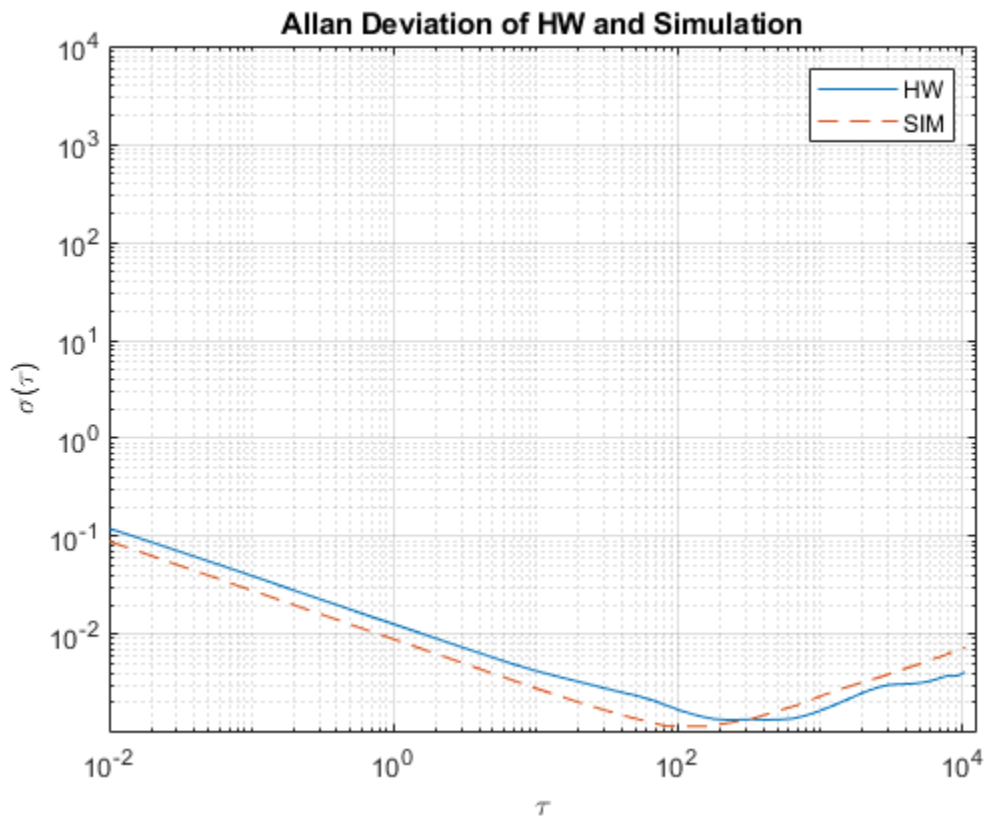
if generateSimulatedData
    % Set the gyroscope parameters to the noise parameters determined
    % above.
    gyro = gyroparams('NoiseDensity', N, 'RandomWalk', K, ...
        'BiasInstability', B);
    omegaSim = helperAllanVarianceExample(L, Fs, gyro);
else
    load('SimulatedSingleAxisGyroscope', 'omegaSim')
end

```

Calculate the simulated Allan deviation and compare it to the logged data.

```
[avarSim, tauSim] = allanvar(omegaSim, 'octave', Fs);
adevSim = sqrt(avarSim);
adevSim = mean(adevSim, 2); % Use the mean of the simulations.
```

```
figure
loglog(tau, adev, tauSim, adevSim, '--')
title('Allan Deviation of HW and Simulation')
xlabel('\tau');
ylabel('\sigma(\tau)')
legend('HW', 'SIM')
grid on
axis equal
```



The plot shows that the gyroscope model created from the `imuSensor` generates measurements with similar Allan deviation to the logged data. The model measurements contain slightly less noise since the quantization and temperature-related parameters are not set using `gyroparams`. The gyroscope model can be used to generate measurements using movements that are not easily captured with hardware.

References

- IEEE Std. 647-2006 IEEE Standard Specification Format Guide and Test Procedure for Single-Axis Laser Gyros

Simulate Inertial Sensor Readings from a Driving Scenario

Generate synthetic sensor data from IMU, GPS, and wheel encoders using driving scenario generation tools from Automated Driving Toolbox™. The `drivingScenario` object simulates the driving scenario and sensor data is generated from the `imuSensor`, `gpsSensor` and `wheelEncoderAckermann` objects.

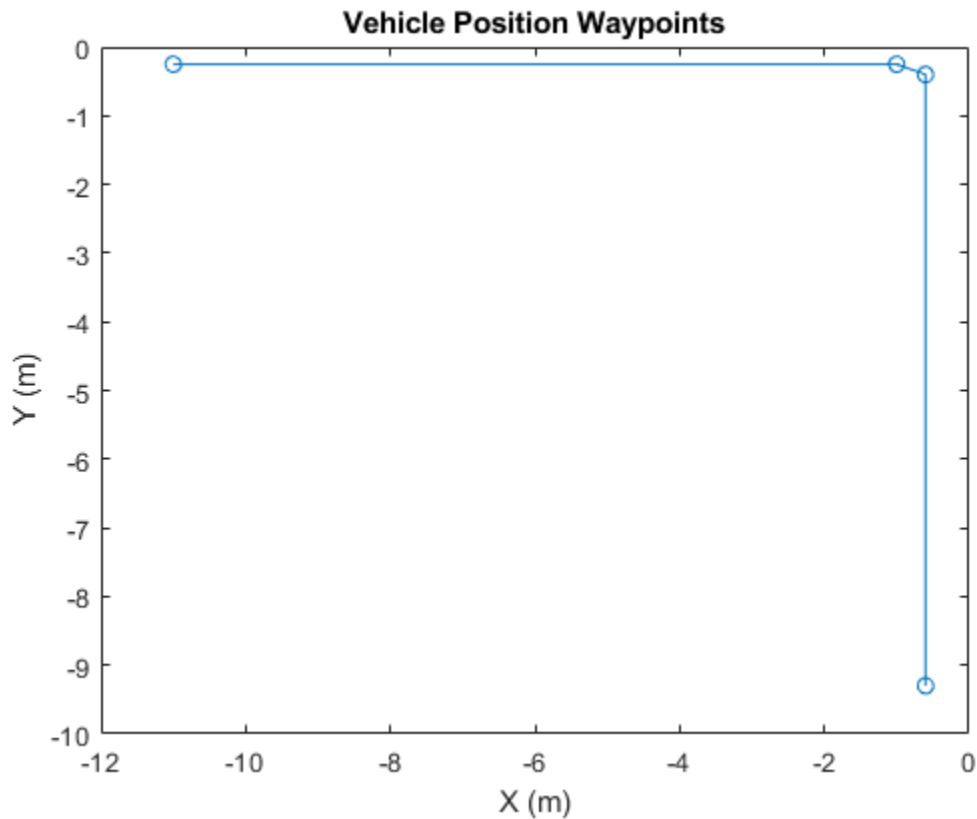
Define Scenario

Create a driving scenario with one vehicle. Define waypoints to have the vehicle move forward and make a turn. To simulate GPS readings, specify the reference location in geodetic coordinates. Generate the vehicle trajectory with the `smoothTrajectory` object function. Plot the waypoints.

```
lla0 = [42 -71 50];
s = drivingScenario('GeoReference',lla0);
v = vehicle(s);

waypoints = [-11 -0.25 0;
             -1 -0.25 0;
             -0.6 -0.4 0;
             -0.6 -9.3 0];
speed = [1.5;0;0.5;1.5];
smoothTrajectory(v,waypoints,speed);

figure
plot(waypoints(:,1),waypoints(:,2),'-o')
xlabel('X (m)')
ylabel('Y (m)')
title('Vehicle Position Waypoints')
```



Create Sensors

Create the IMU, GPS, and wheel encoder sensors. Specify the offset location and angles of the IMU and GPS. These can be edited to change the mounting position and orientation of the sensors on the vehicle.

```
mountingLocationIMU = [1 2 3];
mountingAnglesIMU = [0 0 0];
% Convert orientation offset from Euler angles to quaternion.
orientVeh2IMU = quaternion(mountingAnglesIMU,'eulerd','ZYX','frame');
% ReferenceFrame must be specified as ENU.
imu = imuSensor('SampleRate',1/s.SampleTime,'ReferenceFrame','ENU');

mountingLocationGPS = [1 2 3];
mountingAnglesGPS = [50 40 30];
% Convert orientation offset from Euler angles to quaternion.
orientVeh2GPS = quaternion(mountingAnglesGPS,'eulerd','ZYX','frame');
% The GeoReference property in drivingScenario is equivalent to
% the ReferenceLocation property in gpsSensor.
% ReferenceFrame must be specified as ENU.
gps = gpsSensor('ReferenceLocation',lla0,'ReferenceFrame','ENU');

encoder = wheelEncoderAckermann('TrackWidth',v.Width,...
    'WheelBase',v.Wheelbase,'SampleRate',1/s.SampleTime);
```

Run Simulation

Run the simulation and log the generated sensor readings.

```

% IMU readings.
accel = [];
gyro = [];
% Wheel encoder readings.
ticks = [];
% GPS readings.
lla = [];
gpsVel = [];
% Define the rate of the GPS compared to the simulation rate.
simSamplesPerGPS = (1/s.SampleTime)/gps.SampleRate;
idx = 0;
while advance(s)
    groundTruth = state(v);

    % Unpack the ground truth struct by converting the orientations from
    % Euler angles to quaternions and converting angular velocities from
    % degrees per second to radians per second.
    posVeh = groundTruth.Position;
    orientVeh = quaternion(fliplr(groundTruth.Orientation), 'eulerd', 'ZYX', 'frame');
    velVeh = groundTruth.Velocity;
    accVeh = groundTruth.Acceleration;
    angvelVeh = deg2rad(groundTruth.AngularVelocity);

    % Convert motion quantities from vehicle frame to IMU frame.
    [posIMU,orientIMU,velIMU,accIMU,angvelIMU] = transformMotion( ...
        mountingLocationIMU,orientVeh2IMU, ...
        posVeh,orientVeh,velVeh,accVeh,angvelVeh);
    [accel(end+1,:), gyro(end+1,:)] = imu(accIMU,angvelIMU,orientIMU);

    ticks(end+1,:) = encoder(velVeh, angvelVeh, orientVeh);

    % Only generate a new GPS sample when the simulation has advanced
    % enough.
    if (mod(idx, simSamplesPerGPS) == 0)
        % Convert motion quantities from vehicle frame to GPS frame.
        [posGPS,orientGPS,velGPS,accGPS,angvelGPS] = transformMotion(...
            mountingLocationGPS, orientVeh2GPS,...
            posVeh,orientVeh,velVeh,accVeh,angvelVeh);
        [lla(end+1,:), gpsVel(end+1,:)] = gps(posGPS,velGPS);
    end
    idx = idx + 1;
end

```

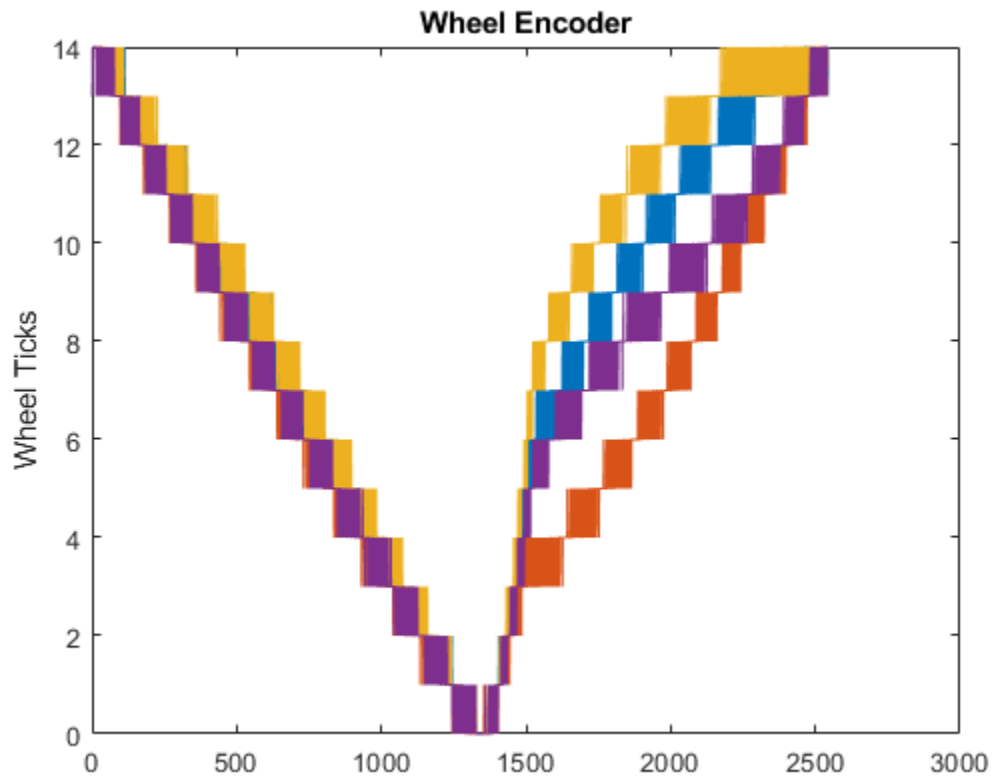
Visualize Results

Plot the generated sensor readings.

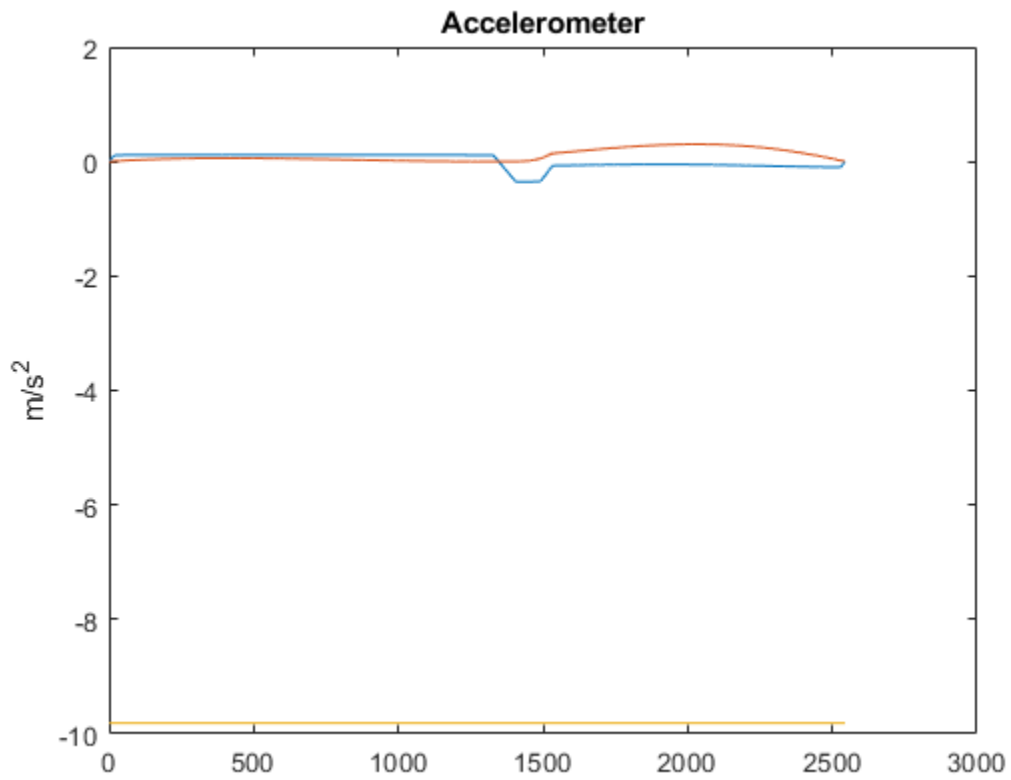
```

figure
plot(ticks)
ylabel('Wheel Ticks')
title('Wheel Encoder')

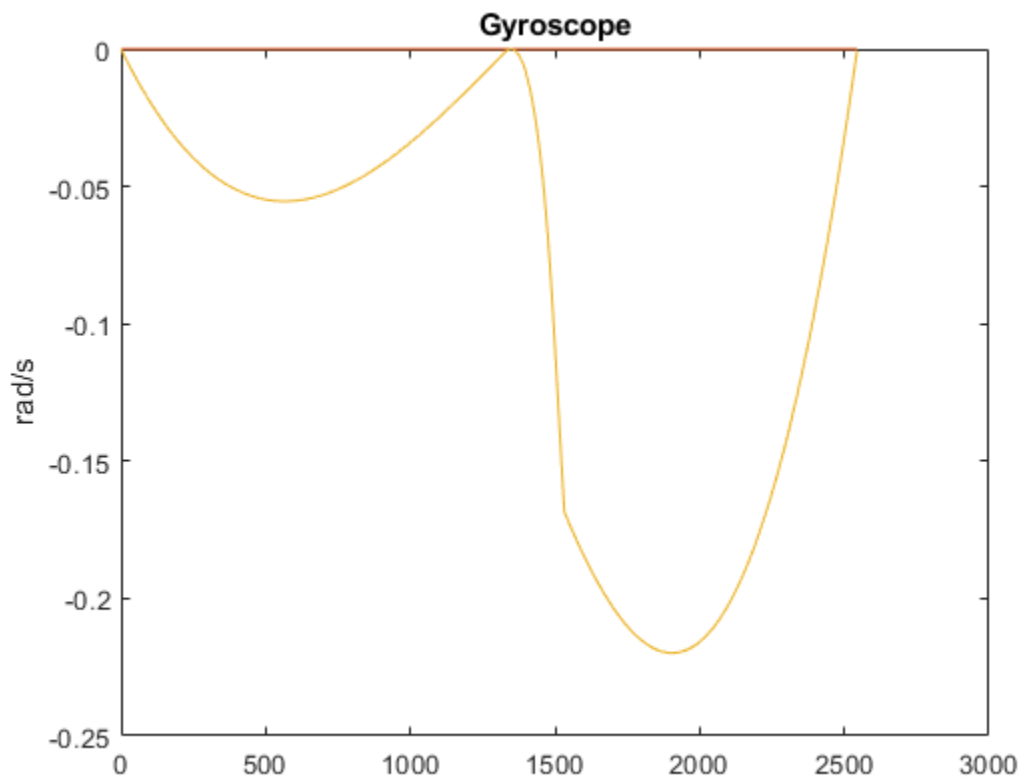
```



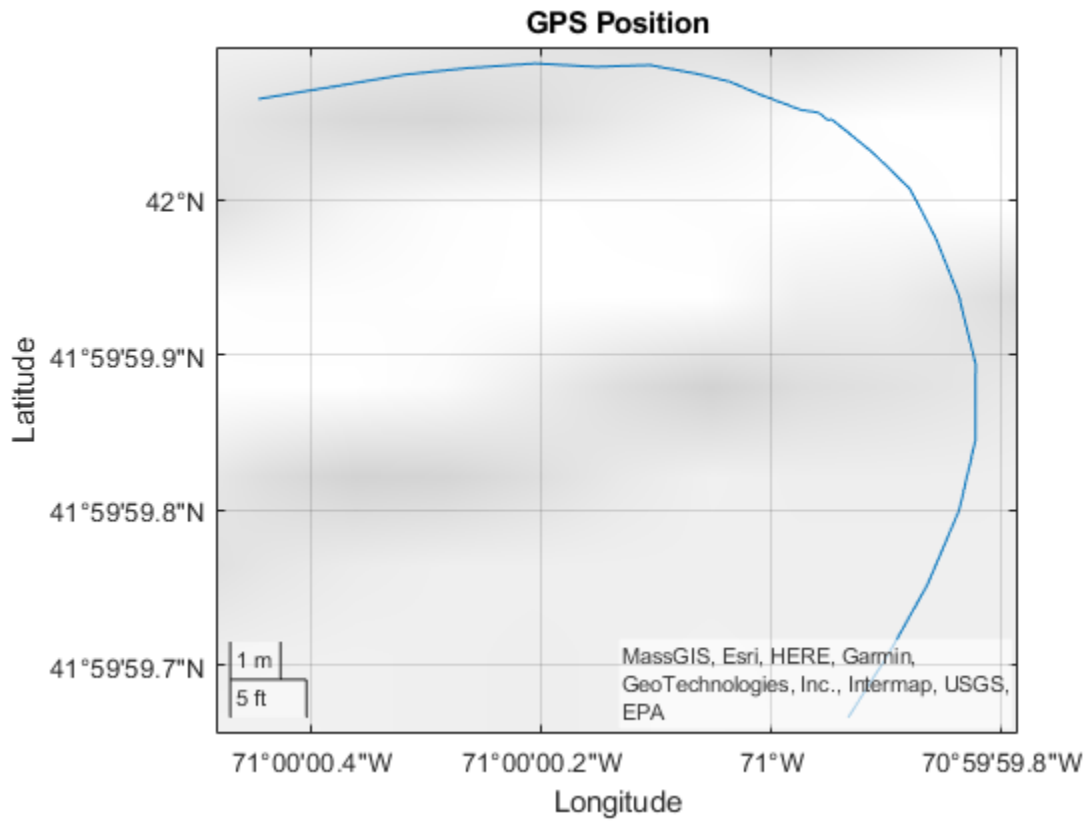
```
figure  
plot(accel)  
ylabel('m/s^2')  
title('Accelerometer')
```



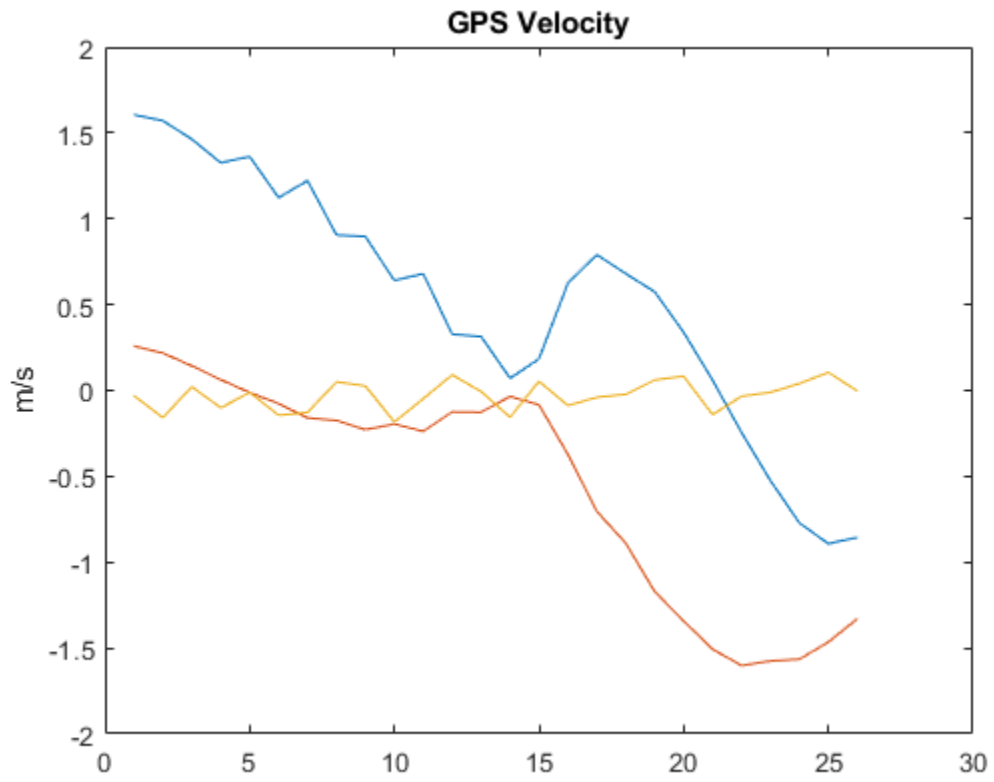
```
figure
plot(gyro)
ylabel('rad/s')
title('Gyroscope')
```



```
figure  
geoplot(lla(:,1),lla(:,2))  
title('GPS Position')
```

```
figure
plot(gpsVel)
ylabel('m/s')
title('GPS Velocity')
```



Estimate Orientation Through Inertial Sensor Fusion

This example shows how to use 6-axis and 9-axis fusion algorithms to compute orientation. There are several algorithms to compute orientation from inertial measurement units (IMUs) and magnetic-angular rate-gravity (MARG) units. This example covers the basics of orientation and how to use these algorithms.

Orientation

An object's orientation describes its rotation relative to some coordinate system, sometimes called a parent coordinate system, in three dimensions.

For the following algorithms, the fixed, parent coordinate system used is North-East-Down (NED). NED is sometimes referred to as the global coordinate system or reference frame. In the NED reference frame, the X-axis points north, the Y-axis points east, and the Z-axis points downward. The X-Y plane of NED is considered to be the local tangent plane of the Earth. Depending on the algorithm, north may be either magnetic north or true north. The algorithms in this example use magnetic north.

If specified, the following algorithms can estimate orientation relative to East-North-Up (ENU) parent coordinate system instead of NED.

An object can be thought of as having its own coordinate system, often called the local or child coordinate system. This child coordinate system rotates with the object relative to the parent coordinate system. If there is no translation, the origins of both coordinate systems overlap.

The orientation quantity computed is a rotation that takes quantities from the parent reference frame to the child reference frame. The rotation is represented by a quaternion or rotation matrix.

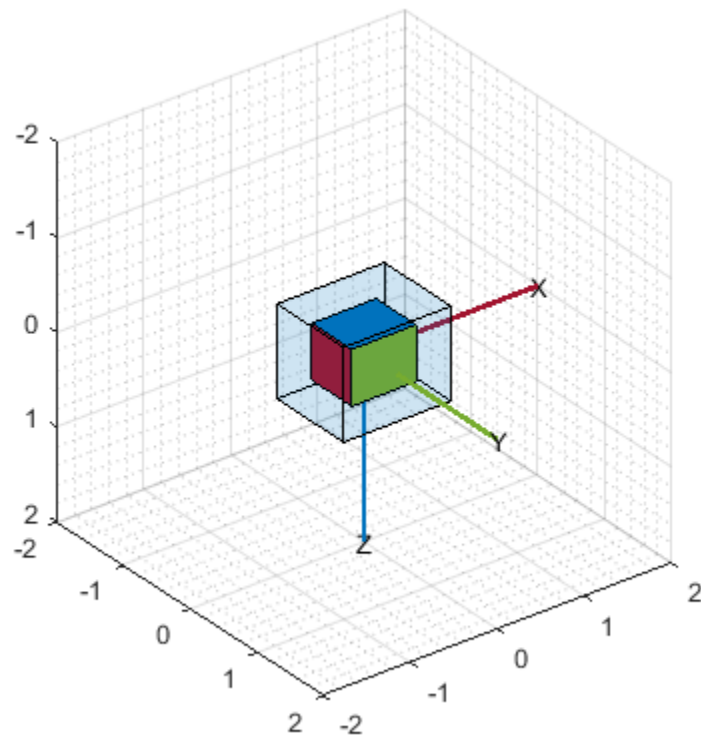
Types of Sensors

For orientation estimation, three types of sensors are commonly used: accelerometers, gyroscopes and magnetometers. Accelerometers measure proper acceleration. Gyroscopes measure angular velocity. Magnetometers measure the local magnetic field. Different algorithms are used to fuse different combinations of sensors to estimate orientation.

Sensor Data

Through most of this example, the same set of sensor data is used. Accelerometer, gyroscope, and magnetometer sensor data was recorded while a device rotated around three different axes: first around its local Y-axis, then around its Z-axis, and finally around its X-axis. The device's X-axis was generally pointed southward for the duration of the experiment.

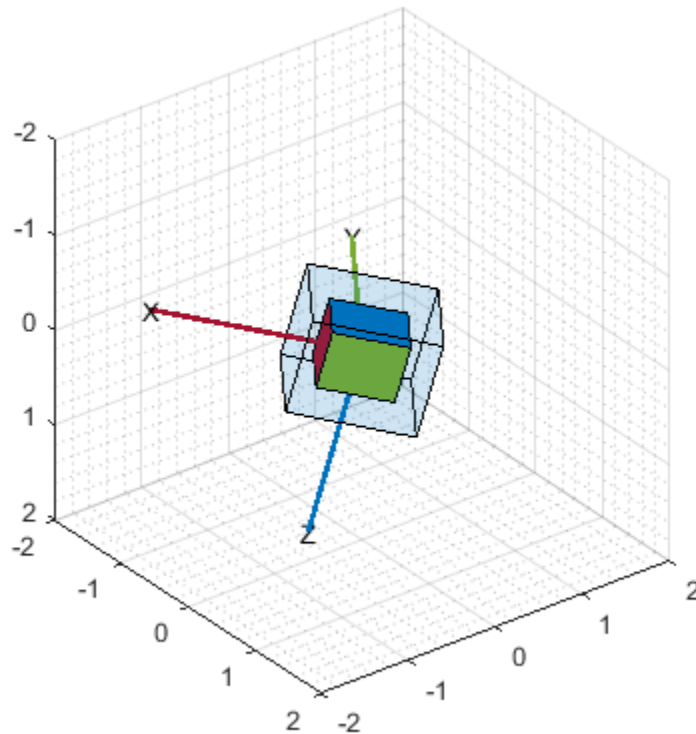
```
ld = load('rpy_9axis.mat');  
  
acc = ld.sensorData.Acceleration;  
gyro = ld.sensorData.AngularVelocity;  
mag = ld.sensorData.MagneticField;  
  
pp = poseplot;
```



Accelerometer-Magnetometer Fusion

The `ecompass` function fuses accelerometer and magnetometer data. This is a memoryless algorithm that requires no parameter tuning, but the algorithm is highly susceptible to sensor noise.

```
qe = ecompass(acc, mag);  
for ii=1:size(acc,1)  
    set(pp, "Orientation", qe(ii))  
    drawnow limitrate  
end
```



Note that the `ecompass` algorithm correctly finds the location of north. However, because the function is memoryless, the estimated motion is not smooth. The algorithm could be used as an initialization step in an orientation filter or some of the techniques presented in the “Lowpass Filter Orientation Using Quaternion SLERP” (Sensor Fusion and Tracking Toolbox) could be used to smooth the motion.

Accelerometer-Gyroscope Fusion

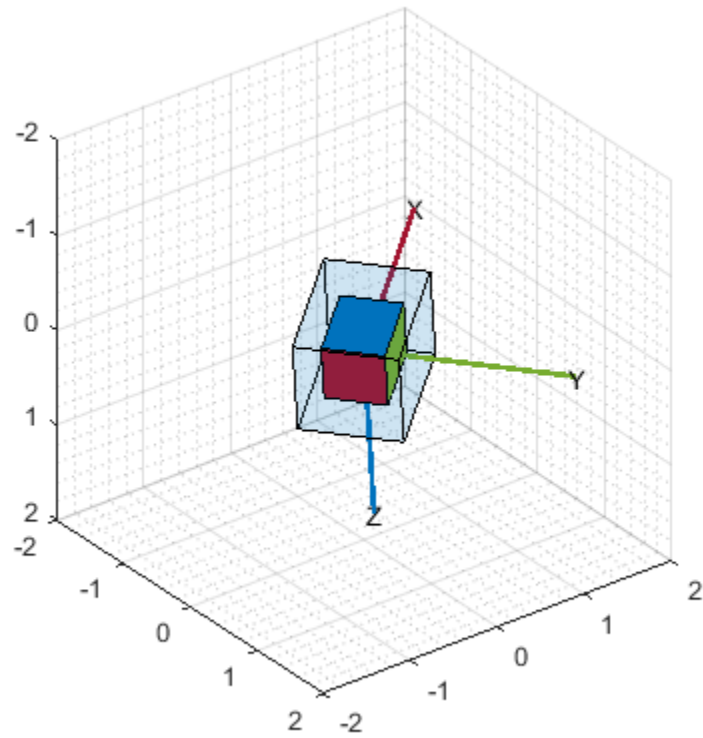
The following objects estimate orientation using either an error-state Kalman filter or a complementary filter. The error-state Kalman filter is the standard estimation filter and allows for many different aspects of the system to be tuned using the corresponding noise parameters. The complementary filter can be used as a substitute for systems with memory constraints, and has minimal tunable parameters, which allows for easier configuration at the cost of finer tuning.

The `imufilter` and `complementaryFilter` System objects™ fuse accelerometer and gyroscope data. The `imufilter` uses an internal error-state Kalman filter and the `complementaryFilter` uses a complementary filter. The filters are capable of removing the gyroscope's bias noise, which drifts over time.

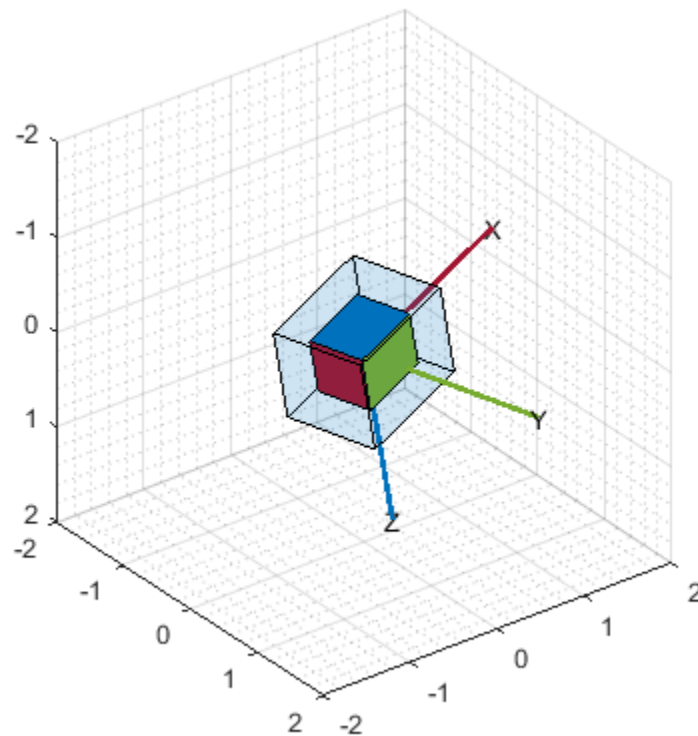
```

ifilt = imufilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qimu = ifilt(acc(ii,:), gyro(ii,:));
    set(pp, "Orientation", qimu)
    drawnow limitrate
end

```



```
% Disable magnetometer input.  
cfilt = complementaryFilter('SampleRate', ld.Fs, 'HasMagnetometer', false);  
for ii=1:size(acc,1)  
    qimu = cfilt(acc(ii,:), gyro(ii,:));  
    set(pp, "Orientation", qimu)  
    drawnow limitrate  
end
```

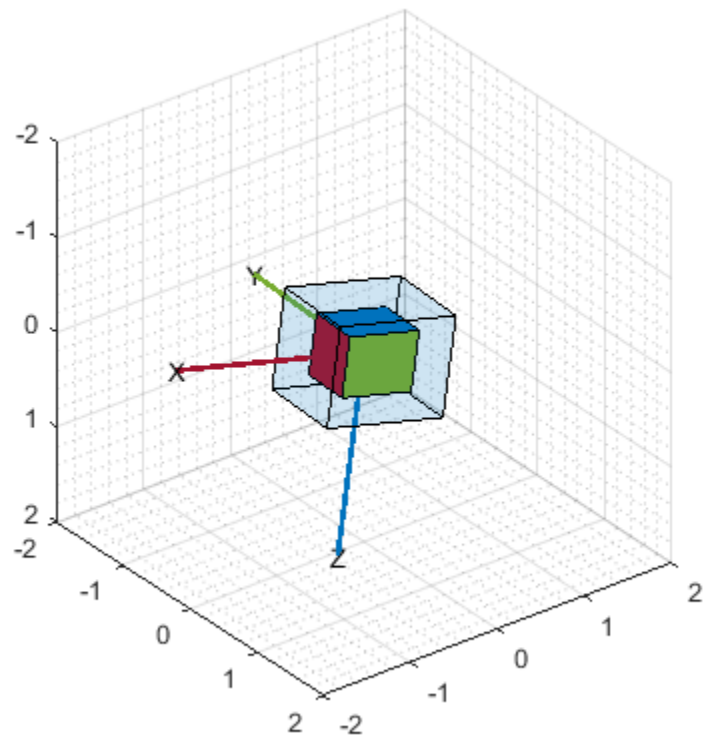


Although the `imufilter` and `complementaryFilter` algorithms produce significantly smoother estimates of the motion, compared to the `ecompass`, they do not correctly estimate the direction of north. The `imufilter` does not process magnetometer data, so it simply assumes the device's X-axis is initially pointing northward. The motion estimate given by `imufilter` is relative to the initial estimated orientation. The `complementaryFilter` makes the same assumption when the `HasMagnetometer` property is set to `false`.

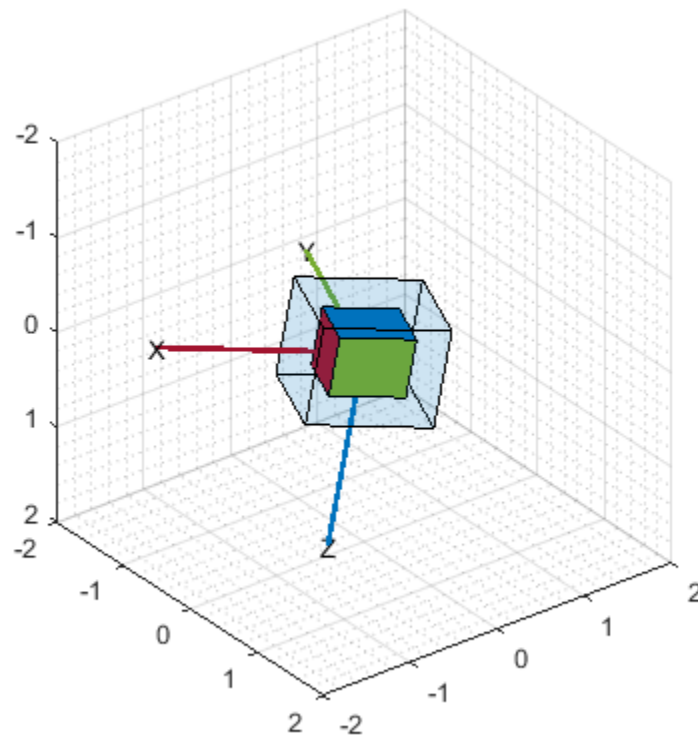
Accelerometer-Gyroscope-Magnetometer Fusion

An attitude and heading reference system (AHRS) consists of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation. The `ahrsfilter` and `complementaryFilter System objects™` combine the best of the previous algorithms to produce a smoothly changing estimate of the device orientation, while correctly estimating the direction of north. The `complementaryFilter` uses the same complementary filter algorithm as before, with an extra step to include the magnetometer and improve the orientation estimate. Like `imufilter`, `ahrsfilter` algorithm also uses an error-state Kalman filter. In addition to gyroscope bias removal, the `ahrsfilter` has some ability to detect and reject mild magnetic jamming.

```
ifilt = ahrsfilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qahrs = ifilt(acc(ii,:), gyro(ii,:), mag(ii,:));
    set(pp, "Orientation", qahrs)
    drawnow limitrate
end
```



```
cfilt = complementaryFilter('SampleRate', ld.Fs);  
for ii=1:size(acc,1)  
    qahrs = cfilt(acc(ii,:), gyro(ii,:), mag(ii,:));  
    set(pp, "Orientation", qahrs)  
    drawnow limitrate  
end
```

Tuning Filter Parameters

The `complementaryFilter`, `imufilter`, and `ahrsfilter` System objects™ all have tunable parameters. Tuning the parameters based on the specified sensors being used can improve performance.

The `complementaryFilter` parameters `AccelerometerGain` and `MagnetometerGain` can be tuned to change the amount each sensor's measurements impact the orientation estimate. When `AccelerometerGain` is set to 0, only the gyroscope is used for the x- and y-axis orientation. When `AccelerometerGain` is set to 1, only the accelerometer is used for the x- and y-axis orientation. When `MagnetometerGain` is set to 0, only the gyroscope is used for the z-axis orientation. When `MagnetometerGain` is set to 1, only the magnetometer is used for the z-axis orientation.

The `ahrsfilter` and `imufilter` System objects™ have more parameters that can allow the filters to more closely match specific hardware sensors. The environment of the sensor is also important to take into account. The `imufilter` parameters are a subset of the `ahrsfilter` parameters. The `AccelerometerNoise`, `GyroscopeNoise`, `MagnetometerNoise`, and `GyroscopeDriftNoise` are measurement noises. The sensors' datasheets help determine those values.

The `LinearAccelerationNoise` and `LinearAccelerationDecayFactor` govern the filter's response to linear (translational) acceleration. Shaking a device is a simple example of adding linear acceleration.

Consider how an `imufilter` with a `LinearAccelerationNoise` of $9e-3 (m/s^2)^2$ responds to a shaking trajectory, compared to one with a `LinearAccelerationNoise` of $9e-4 (m/s^2)^2$.

```

ld = load('shakingDevice.mat');
accel = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;

highVarFilt = imfilter('SampleRate', ld.Fs, ...
    'LinearAccelerationNoise', 0.009);
qHighLANoise = highVarFilt(accel, gyro);

lowVarFilt = imfilter('SampleRate', ld.Fs, ...
    'LinearAccelerationNoise', 0.0009);
qLowLANoise = lowVarFilt(accel, gyro);

```

One way to see the effect of the LinearAccelerationNoise is to look at the output gravity vector. The gravity vector is simply the third column of the orientation rotation matrix.

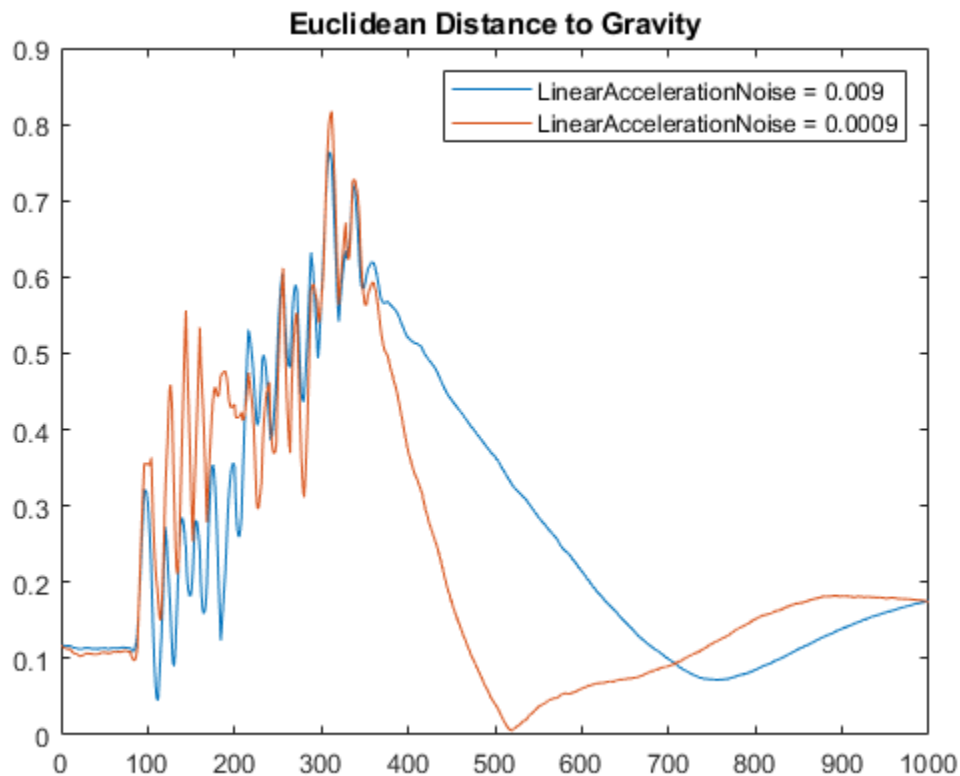
```

rmatHigh = rotmat(qHighLANoise, 'frame');
rmatLow = rotmat(qLowLANoise, 'frame');

gravDistHigh = sqrt(sum( (rmatHigh(:,3,:) - [0;0;1]).^2, 1));
gravDistLow = sqrt(sum( (rmatLow(:,3,:) - [0;0;1]).^2, 1));

figure;
plot([squeeze(gravDistHigh), squeeze(gravDistLow)]);
title('Euclidean Distance to Gravity');
legend('LinearAccelerationNoise = 0.009', ...
    'LinearAccelerationNoise = 0.0009');

```



The `lowVarFilt` has a low `LinearAccelerationNoise`, so it expects to be in an environment with low linear acceleration. Therefore, it is more susceptible to linear acceleration, as illustrated by the large variations earlier in the plot. However, because it expects to be in an environment with a low linear acceleration, higher trust is placed in the accelerometer signal. As such, the orientation estimate converges quickly back to vertical once the shaking has ended. The converse is true for `highVarFilt`. The filter is less affected by shaking, but the orientation estimate takes longer to converge to vertical when the shaking has stopped.

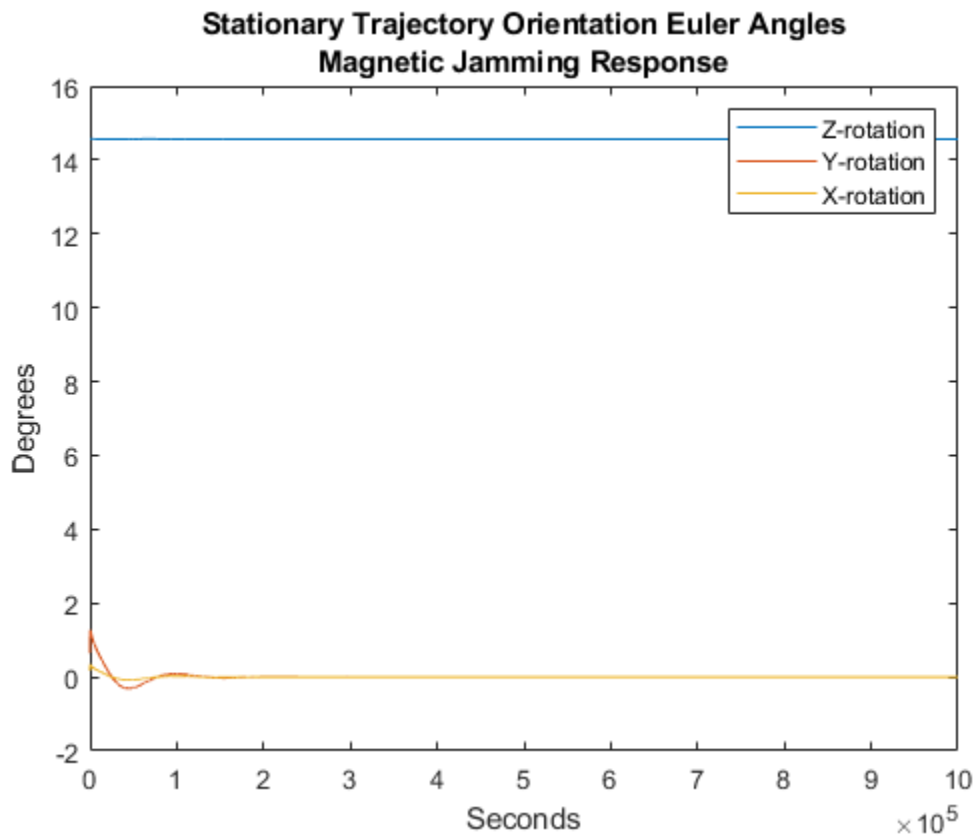
The `MagneticDisturbanceNoise` property enables modeling magnetic disturbances (non-geomagnetic noise sources) in much the same way `LinearAccelerationNoise` models linear acceleration.

The two decay factor properties (`MagneticDisturbanceDecayFactor` and `LinearAccelerationDecayFactor`) model the rate of variation of the noises. For slowly varying noise sources, set these parameters to a value closer to 1. For quickly varying, uncorrelated noises, set these parameters closer to 0. A lower `LinearAccelerationDecayFactor` enables the orientation estimate to find "down" more quickly. A lower `MagneticDisturbanceDecayFactor` enables the orientation estimate to find north more quickly.

Very large, short magnetic disturbances are rejected almost entirely by the `ahrsfilter`. Consider a pulse of [0 250 0] uT applied while recording from a stationary sensor. Ideally, there should be no change in orientation estimate.

```
ld = load('magJamming.mat');
hpulse = ahrsfilter('SampleRate', ld.Fs);
len = 1:10000;
qpulse = hpulse(ld.sensorData.Acceleration(len,:), ...
    ld.sensorData.AngularVelocity(len,:), ...
    ld.sensorData.MagneticField(len,:));

figure;
timevec = 0:ld.Fs:(ld.Fs*numel(qpulse) - 1);
plot( timevec, eulerd(qpulse, 'ZYX', 'frame') );
title(['Stationary Trajectory Orientation Euler Angles' newline ...
    'Magnetic Jamming Response']);
legend('Z-rotation', 'Y-rotation', 'X-rotation');
ylabel('Degrees');
xlabel('Seconds');
```



Note that the filter almost totally rejects this magnetic pulse as interference. Any magnetic field strength greater than four times the `ExpectedMagneticFieldStrength` is considered a jamming source and the magnetometer signal is ignored for those samples.

Conclusion

The algorithms presented here, when properly tuned, enable estimation of orientation and are robust against environmental noise sources. It is important to consider the situations in which the sensors are used and tune the filters accordingly.

IMU and GPS Fusion for Inertial Navigation

This example shows how you might build an IMU + GPS fusion algorithm suitable for unmanned aerial vehicles (UAVs) or quadcopters.

This example uses accelerometers, gyroscopes, magnetometers, and GPS to determine orientation and position of a UAV.

Simulation Setup

Set the sampling rates. In a typical system, the accelerometer and gyroscope run at relatively high sample rates. The complexity of processing data from those sensors in the fusion algorithm is relatively low. Conversely, the GPS, and in some cases the magnetometer, run at relatively low sample rates, and the complexity associated with processing them is high. In this fusion algorithm, the magnetometer and GPS samples are processed together at the same low rate, and the accelerometer and gyroscope samples are processed together at the same high rate.

To simulate this configuration, the IMU (accelerometer, gyroscope, and magnetometer) are sampled at 160 Hz, and the GPS is sampled at 1 Hz. Only one out of every 160 samples of the magnetometer is given to the fusion algorithm, so in a real system the magnetometer could be sampled at a much lower rate.

```
imuFs = 160;
gpsFs = 1;

% Define where on the Earth this simulated scenario takes place using the
% latitude, longitude and altitude.
refloc = [42.2825 -72.3430 53.0352];

% Validate that the |gpsFs| divides |imuFs|. This allows the sensor sample
% rates to be simulated using a nested for loop without complex sample rate
% matching.

imuSamplesPerGPS = (imuFs/gpsFs);
assert(imuSamplesPerGPS == fix(imuSamplesPerGPS), ...
    'GPS sampling rate must be an integer factor of IMU sampling rate.');
```

Fusion Filter

Create the filter to fuse IMU + GPS measurements. The fusion filter uses an extended Kalman filter to track orientation (as a quaternion), velocity, position, sensor biases, and the geomagnetic vector.

This `insfilterMARG` has a few methods to process sensor data, including `predict`, `fusemag` and `fusegps`. The `predict` method takes the accelerometer and gyroscope samples from the IMU as inputs. Call the `predict` method each time the accelerometer and gyroscope are sampled. This method predicts the states one time step ahead based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated here.

The `fusegps` method takes GPS samples as input. This method updates the filter states based on GPS samples by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated here, this time using the Kalman gain as well.

The `fusemag` method is similar but updates the states, Kalman gain, and error covariance based on the magnetometer samples.

Though the `insfilterMARG` takes accelerometer and gyroscope samples as inputs, these are integrated to compute delta velocities and delta angles, respectively. The filter tracks the bias of the magnetometer and these integrated signals.

```
fusionfilt = insfilterMARG;
fusionfilt.IMUSampleRate = imuFs;
fusionfilt.ReferenceLocation = refloc;
```

UAV Trajectory

This example uses a saved trajectory recorded from a UAV as the ground truth. This trajectory is fed to several sensor simulators to compute simulated accelerometer, gyroscope, magnetometer, and GPS data streams.

```
% Load the "ground truth" UAV trajectory.
load LoggedQuadcopter.mat trajData;
trajOrient = trajData.Orientation;
trajVel = trajData.Velocity;
trajPos = trajData.Position;
trajAcc = trajData.Acceleration;
trajAngVel = trajData.AngularVelocity;

% Initialize the random number generator used in the simulation of sensor
% noise.
rng(1)
```

GPS Sensor

Set up the GPS at the specified sample rate and reference location. The other parameters control the nature of the noise in the output signal.

```
gps = gpsSensor('UpdateRate', gpsFs);
gps.ReferenceLocation = refloc;
gps.DecayFactor = 0.5; % Random walk noise parameter
gps.HorizontalPositionAccuracy = 1.6;
gps.VerticalPositionAccuracy = 1.6;
gps.VelocityAccuracy = 0.1;
```

IMU Sensors

Typically, a UAV uses an integrated MARG sensor (Magnetic, Angular Rate, Gravity) for pose estimation. To model a MARG sensor, define an IMU sensor model containing an accelerometer, gyroscope, and magnetometer. In a real-world application the three sensors could come from a single integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors.

```
imu = imuSensor('accel-gyro-mag', 'SampleRate', imuFs);
imu.MagneticField = [19.5281 -5.0741 48.0067];

% Accelerometer
imu.Accelerometer.MeasurementRange = 19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.ConstantBias = 0.19;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
```

```
imu.Gyroscope.ConstantBias = deg2rad(3.125);
imu.Gyroscope.AxesMisalignment = 1.5;
imu.Gyroscope.NoiseDensity = deg2rad(0.025);

% Magnetometer
imu.Magnetometer.MeasurementRange = 1000;
imu.Magnetometer.Resolution = 0.1;
imu.Magnetometer.ConstantBias = 100;
imu.Magnetometer.NoiseDensity = 0.3/ sqrt(50);
```

Initialize the State Vector of the insfilterMARG

The insfilterMARG tracks the pose states in a 22-element vector. The states are:

State	Units	State Vector Index
Orientation as a quaternion		1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Delta Angle Bias (XYZ)	rad	11:13
Delta Velocity Bias (XYZ)	m/s	14:16
Geomagnetic Field Vector (NED)	uT	17:19
Magnetometer Bias (XYZ)	uT	20:22

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
% Initialize the states of the filter

initstate = zeros(22,1);
initstate(1:4) = compact( meanrot(trajOrient(1:100)));
initstate(5:7) = mean( trajPos(1:100,:), 1);
initstate(8:10) = mean( trajVel(1:100,:), 1);
initstate(11:13) = imu.Gyroscope.ConstantBias./imuFs;
initstate(14:16) = imu.Accelerometer.ConstantBias./imuFs;
initstate(17:19) = imu.MagneticField;
initstate(20:22) = imu.Magnetometer.ConstantBias;

fusionfilt.State = initstate;
```

Initialize the Variances of the insfilterMARG

The insfilterMARG measurement noises describe how much noise is corrupting the sensor reading. These values are based on the imuSensor and gpsSensor parameters.

The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter.

```
% Measurement noises
Rmag = 0.09; % Magnetometer measurement noise
Rvel = 0.01; % GPS Velocity measurement noise
Rpos = 2.56; % GPS Position measurement noise

% Process noises
fusionfilt.AccelerometerBiasNoise = 2e-4;
fusionfilt.AccelerometerNoise = 2;
fusionfilt.GyroscopeBiasNoise = 1e-16;
fusionfilt.GyroscopeNoise = 1e-5;
fusionfilt.MagnetometerBiasNoise = 1e-10;
```

```
fusionfilt.GeoMagneticVectorNoise = 1e-12;

% Initial error covariance
fusionfilt.StateCovariance = 1e-9*ones(22);
```

Initialize Scopes

The `HelperScrollingPlotter` scope enables plotting of variables over time. It is used here to track errors in pose. The `HelperPoseViewer` scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to false.

```
useErrScope = true; % Turn on the streaming error plot
usePoseView = true; % Turn on the 3-D pose viewer
```

```
if useErrScope
    errslope = HelperScrollingPlotter(...
        'NumInputs', 4, ...
        'TimeSpan', 10, ...
        'SampleRate', imuFs, ...
        'YLabel', {'degrees', ...
        'meters', ...
        'meters', ...
        'meters'}, ...
        'Title', {'Quaternion Distance', ...
        'Position X Error', ...
        'Position Y Error', ...
        'Position Z Error'}, ...
        'YLimits', ...
        [-1, 1
        -2, 2
        -2 2
        -2 2]);
end
```

```
if usePoseView
    posescope = HelperPoseViewer(...
        'XPositionLimits', [-15 15], ...
        'YPositionLimits', [-15, 15], ...
        'ZPositionLimits', [-10 10]);
end
```

Simulation Loop

The main simulation loop is a while loop with a nested for loop. The while loop executes at `gpsFs`, which is the GPS sample rate. The nested for loop executes at `imuFs`, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

```
% Loop setup - |trajData| has about 142 seconds of recorded data.
secondsToSimulate = 50; % simulate about 50 seconds
numsamples = secondsToSimulate*imuFs;
```

```
loopBound = floor(numsamples);
loopBound = floor(loopBound/imuFs)*imuFs; % ensure enough IMU Samples
```

```
% Log data for final metric computation.
pqorient = quaternion.zeros(loopBound,1);
pqpos = zeros(loopBound,3);
```



```

fcnt = 1;
while(fcnt <=loopBound)
    % |predict| loop at IMU update frequency.
    for ff=1:imuSamplesPerGPS
        % Simulate the IMU data from the current pose.
        [accel, gyro, mag] = imu(trajAcc(fcnt,:), trajAngVel(fcnt, :), ...
            trajOrient(fcnt));

        % Use the |predict| method to estimate the filter state based
        % on the simulated accelerometer and gyroscope signals.
        predict(fusionfilt, accel, gyro);

        % Acquire the current estimate of the filter states.
        [fusedPos, fusedOrient] = pose(fusionfilt);

        % Save the position and orientation for post processing.
        pqorient(fcnt) = fusedOrient;
        pqpos(fcnt,:) = fusedPos;

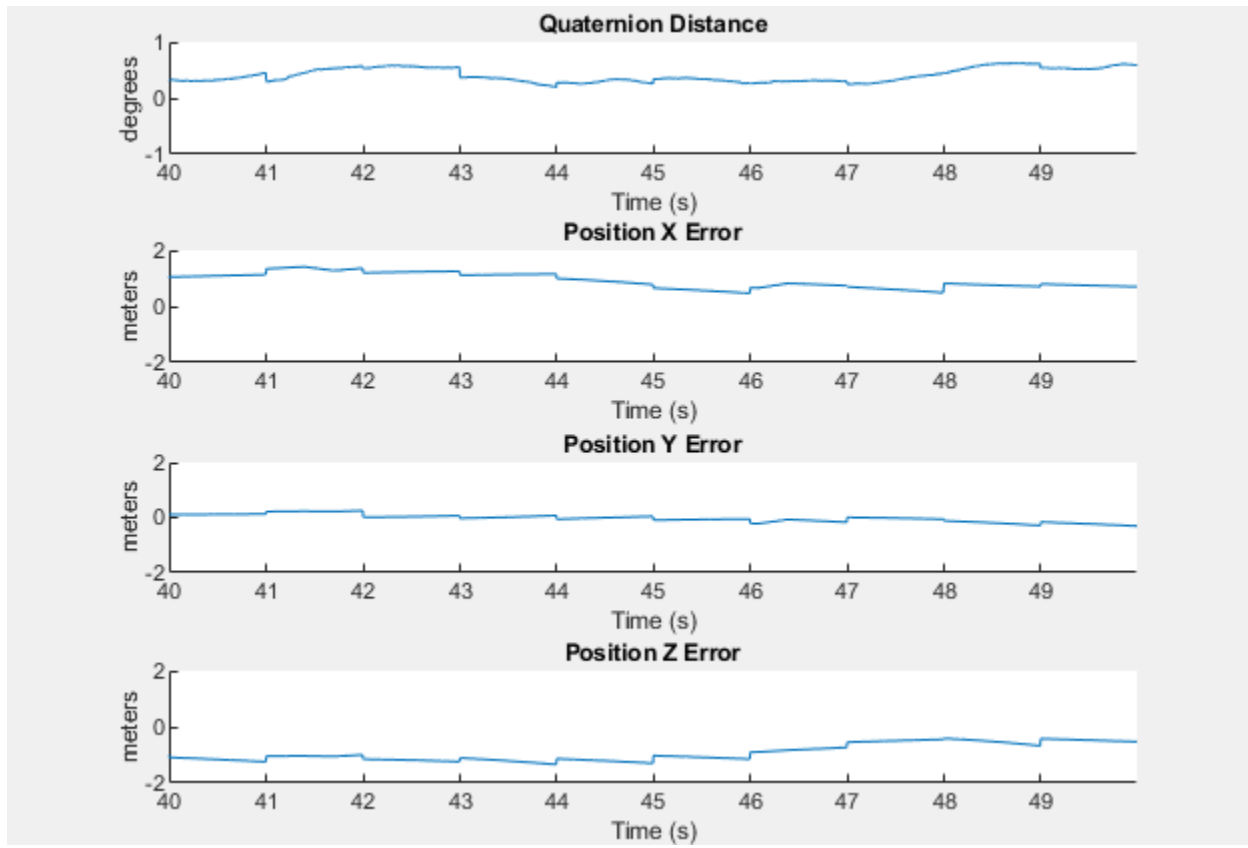
        % Compute the errors and plot.
        if useErrScope
            orientErr = rad2deg(dist(fusedOrient, ...
                trajOrient(fcnt) ));
            posErr = fusedPos - trajPos(fcnt,:);
            errsscope(orientErr, posErr(1), posErr(2), posErr(3));
        end

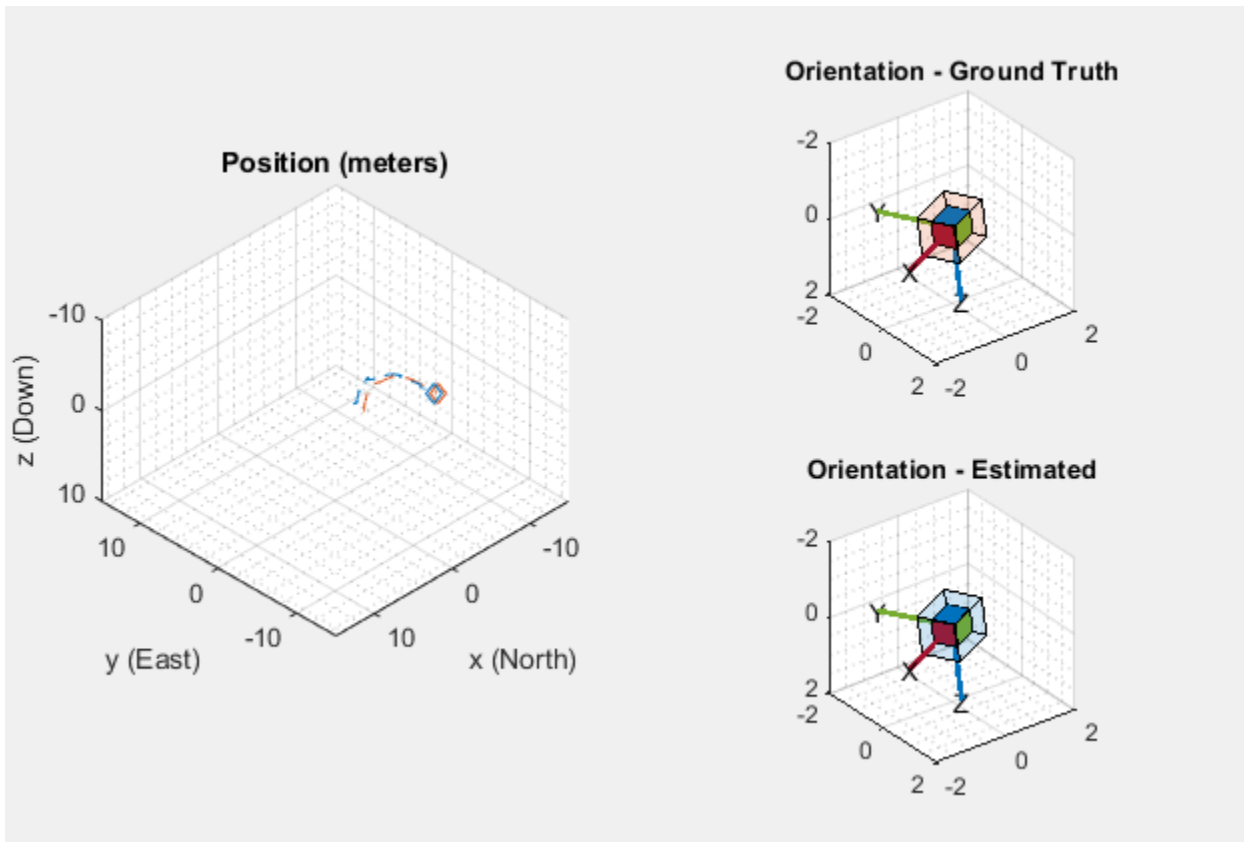
        % Update the pose viewer.
        if usePoseView
            posescope(pqpos(fcnt,:), pqorient(fcnt), trajPos(fcnt,:), ...
                trajOrient(fcnt,:) );
        end
        fcnt = fcnt + 1;
    end

    % This next step happens at the GPS sample rate.
    % Simulate the GPS output based on the current pose.
    [lla, gpsvel] = gps( trajPos(fcnt,:), trajVel(fcnt,:) );

    % Correct the filter states based on the GPS data and magnetic
    % field measurements.
    fusegps(fusionfilt, lla, Rpos, gpsvel, Rvel);
    fusemag(fusionfilt, mag, Rmag);
end

```





Error Metric Computation

Position and orientation estimates were logged throughout the simulation. Now compute an end-to-end root mean squared error for both position and orientation.

```
posd = pqpos(1:loopBound,:) - trajPos( 1:loopBound, :);
```

```
% For orientation, quaternion distance is a much better alternative to
% subtracting Euler angles, which have discontinuities. The quaternion
% distance can be computed with the |dist| function, which gives the
% angular difference in orientation in radians. Convert to degrees
% for display in the command window.
```

```
quadd = rad2deg(dist(pqorient(1:loopBound), trajOrient(1:loopBound)) );
```

```
% Display RMS errors in the command window.
fprintf('\n\nEnd-to-End Simulation Position RMS Error\n');
```

```
End-to-End Simulation Position RMS Error
```

```
msep = sqrt(mean(posd.^2));
fprintf('\tX: %.2f , Y: %.2f, Z: %.2f (meters)\n\n',msep(1), ...
        msep(2), msep(3));
```

```
X: 0.57 , Y: 0.53, Z: 0.68 (meters)
```

```
fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
```

End-to-End Quaternion Distance RMS Error (degrees)

```
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));
```

```
0.32 (degrees)
```

GNSS Simulation Overview

Global Navigation Satellite System (GNSS) simulation generates receiver position estimates. These receiver position estimates come from GPS and GNSS sensor models as `gpsSensor` and `gnssSensor` objects. Monitor the status of the position estimate in the `gnssSensor` using the dilution of precision outputs and compare the number of satellites available.

Simulation Parameters

Specify the parameters to be used in the GNSS simulation:

- Sampling rate of the GNSS receiver
- Local navigation reference frame
- Location on Earth in latitude, longitude, and altitude (LLA) coordinates
- Number of samples to simulate

```
Fs = 1;
refFrame = "NED";
lla0 = [42.2825 -71.343 53.0352];
N = 100;
```

Create a trajectory for a stationary sensor.

```
pos = zeros(N, 3);
vel = zeros(N, 3);
time = (0:N-1) ./ Fs;
```

Sensor Model Creation

Create the GNSS simulation objects, `gpsSensor` and `gnssSensor` using the same initial parameters for each.

```
gps = gpsSensor("SampleRate", Fs, "ReferenceLocation", lla0, ...
    "ReferenceFrame", refFrame);

gnss = gnssSensor("SampleRate", Fs, "ReferenceLocation", lla0, ...
    "ReferenceFrame", refFrame);
```

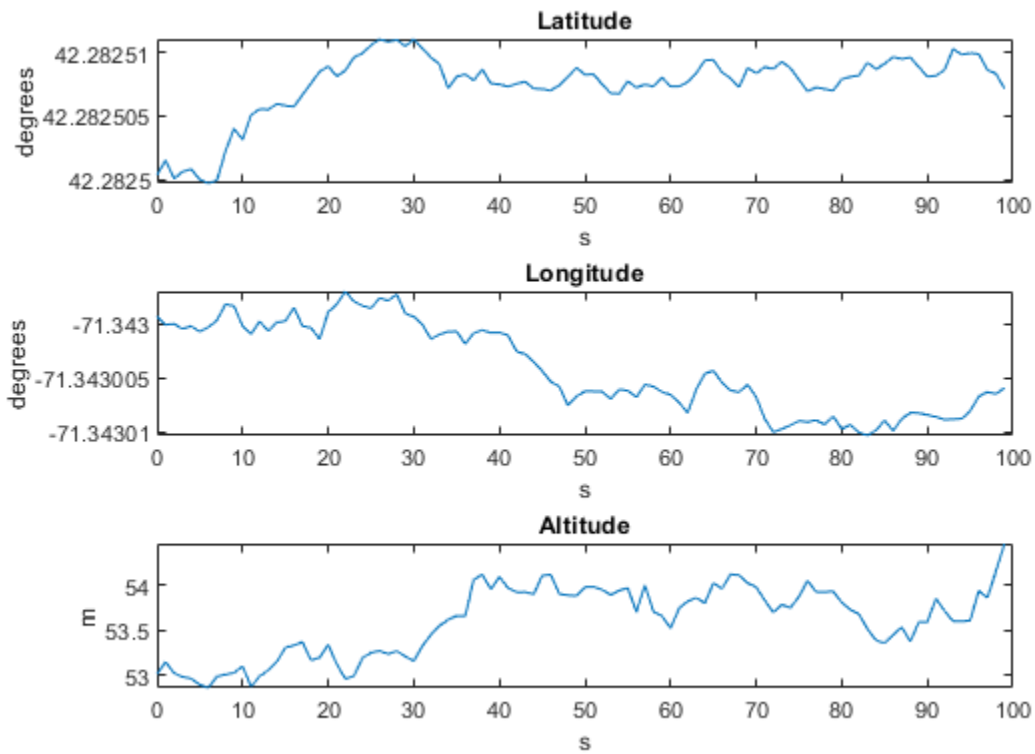
Simulation using `gpsSensor`

Generate outputs from a stationary receiver using the GPS sensor. Visualize the position in LLA coordinates and the velocity in each direction.

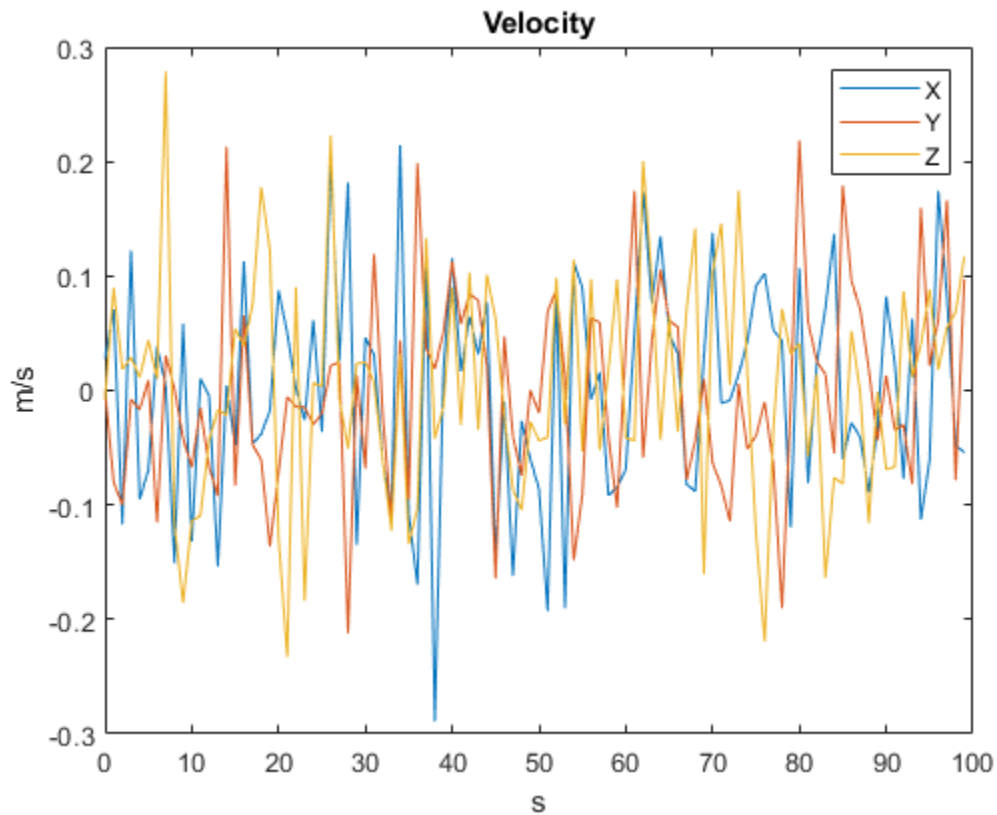
```
% Generate outputs.
[llaGPS, velGPS] = gps(pos, vel);

% Visualize position.
figure
subplot(3, 1, 1)
plot(time, llaGPS(:,1))
title('Latitude')
ylabel('degrees')
xlabel('s')
subplot(3, 1, 2)
plot(time, llaGPS(:,2))
title('Longitude')
```

```
ylabel('degrees')
xlabel('s')
subplot(3, 1, 3)
plot(time, llaGPS(:,3))
title('Altitude')
ylabel('m')
xlabel('s')
```



```
% Visualize velocity.
figure
plot(time, velGPS)
title('Velocity')
legend('X', 'Y', 'Z')
ylabel('m/s')
xlabel('s')
```

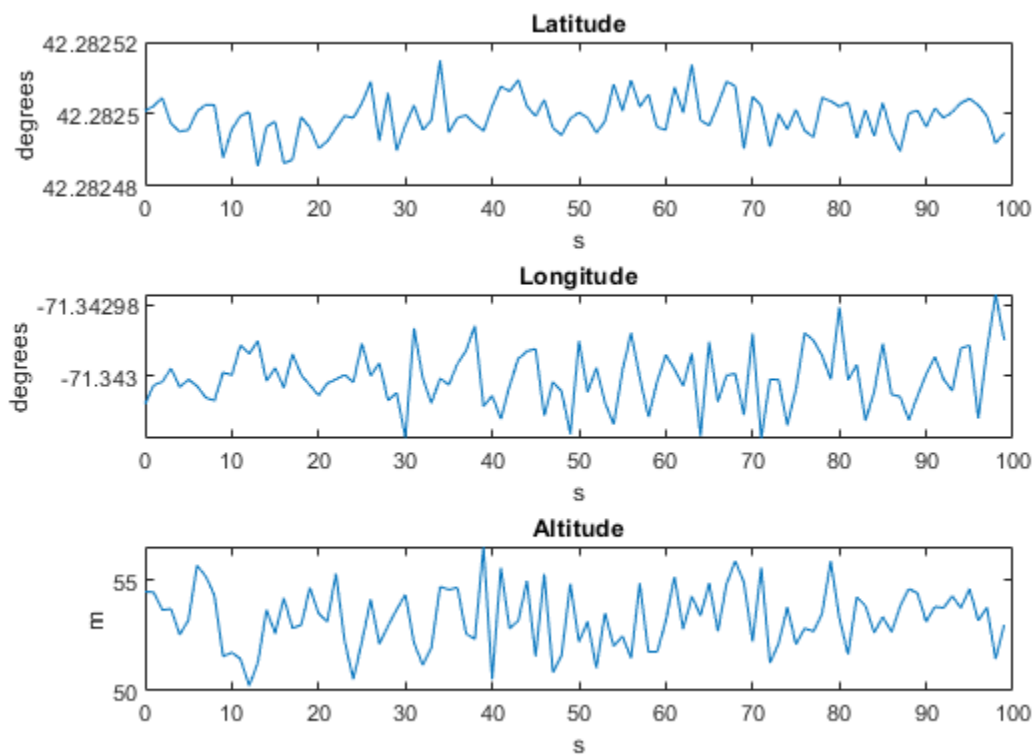


Simulation using gnssSensor

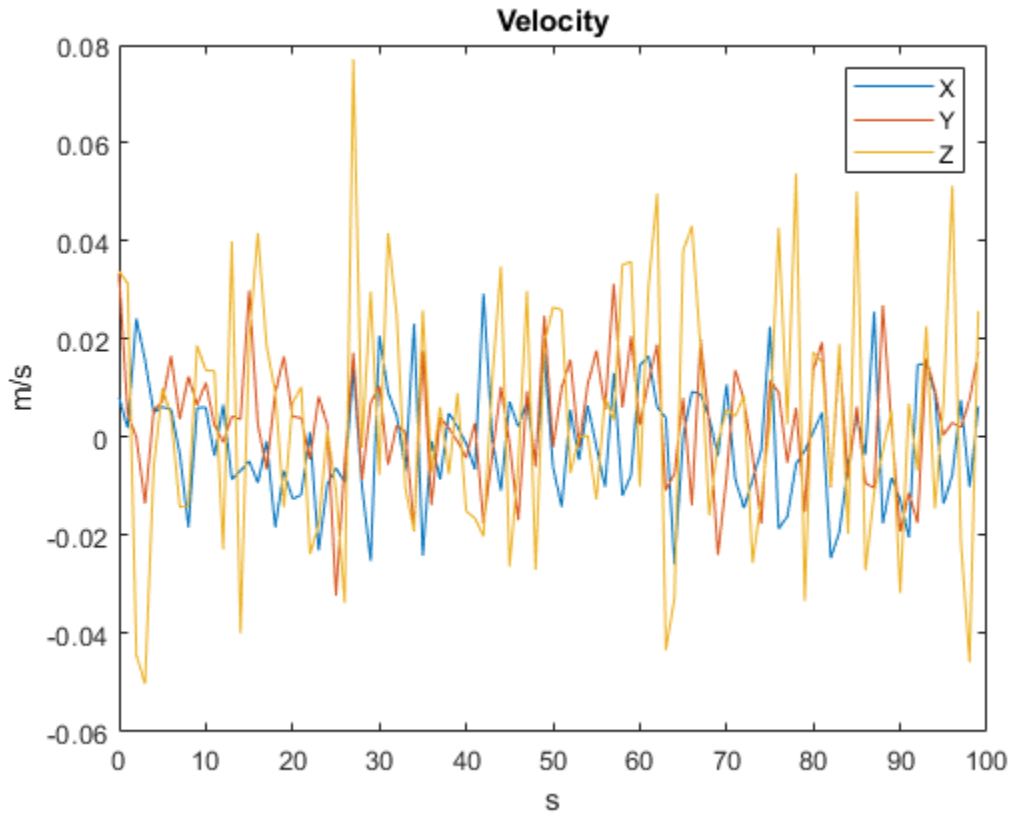
Generate outputs from a stationary receiver using the GNSS sensor. Visualize the position and velocity and notice the differences in the simulation.

```
% Generate outputs.
[llaGNSS, velGNSS] = gnss(pos, vel);

% Visualize positon.
figure
subplot(3, 1, 1)
plot(time, llaGNSS(:,1))
title('Latitude')
ylabel('degrees')
xlabel('s')
subplot(3, 1, 2)
plot(time, llaGNSS(:,2))
title('Longitude')
ylabel('degrees')
xlabel('s')
subplot(3, 1, 3)
plot(time, llaGNSS(:,3))
title('Altitude')
ylabel('m')
xlabel('s')
```



```
% Visualize velocity.  
figure  
plot(time, velGNSS)  
title('Velocity')  
legend('X', 'Y', 'Z')  
ylabel('m/s')  
xlabel('s')
```

Dilution of Precision

The `gnssSensor` object has a higher fidelity simulation compared to `gpsSensor`. For example, the `gnssSensor` object uses simulated satellite positions to estimate the receiver position. This means that the horizontal dilution of precision (HDOP) and vertical dilution of precision (VDOP) can be reported along with the position estimate. These values indicate how precise the position estimate is based on the satellite geometry. Smaller values indicate a more precise estimate.

```
% Set the RNG seed to reproduce results.
rng('default')

% Specify the start time of the simulation.
initTime = datetime(2020, 4, 20, 18, 10, 0, "TimeZone", "America/New_York");

% Create the GNSS receiver model.
gnss = gnssSensor("SampleRate", Fs, "ReferenceLocation", lla0, ...
    "ReferenceFrame", refFrame, "InitialTime", initTime);

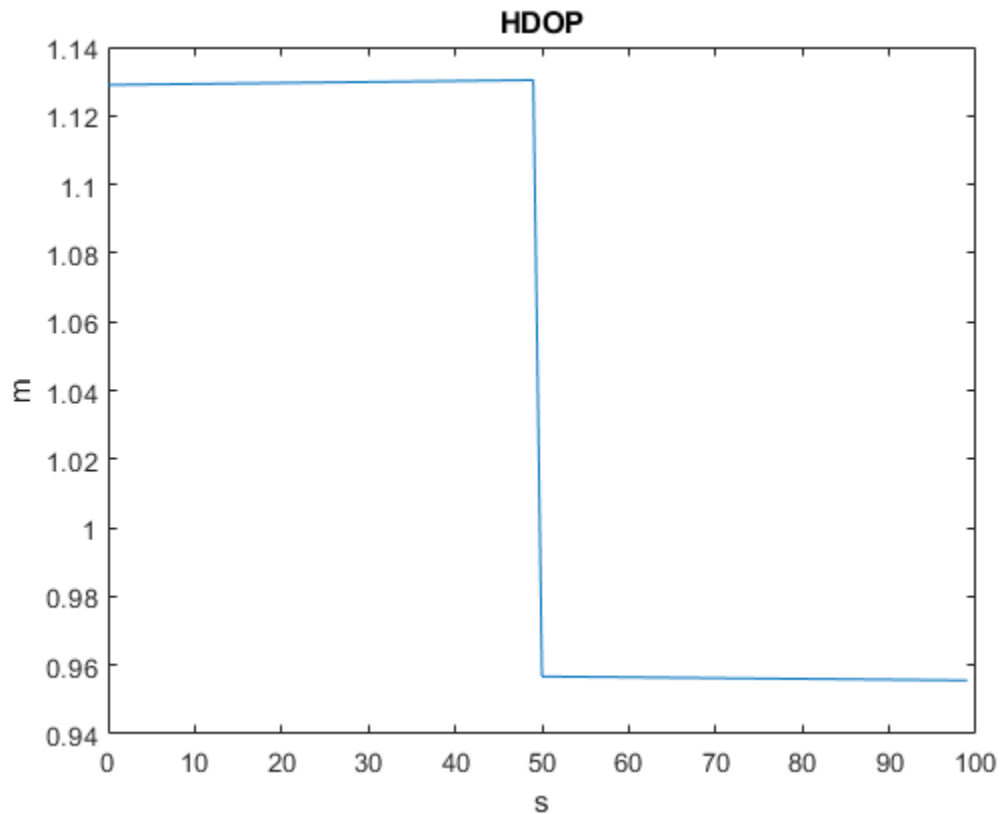
% Obtain the receiver status.
[~, ~, status] = gnss(pos, vel);
disp(status(1))

    SatelliteAzimuth: [7x1 double]
    SatelliteElevation: [7x1 double]
                HDOP: 1.1290
                VDOP: 1.9035
```

View the HDOP throughout the simulation. There is a decrease in the HDOP. This means that the satellite geometry changed.

```
hdops = vertcat(status.HDOP);
```

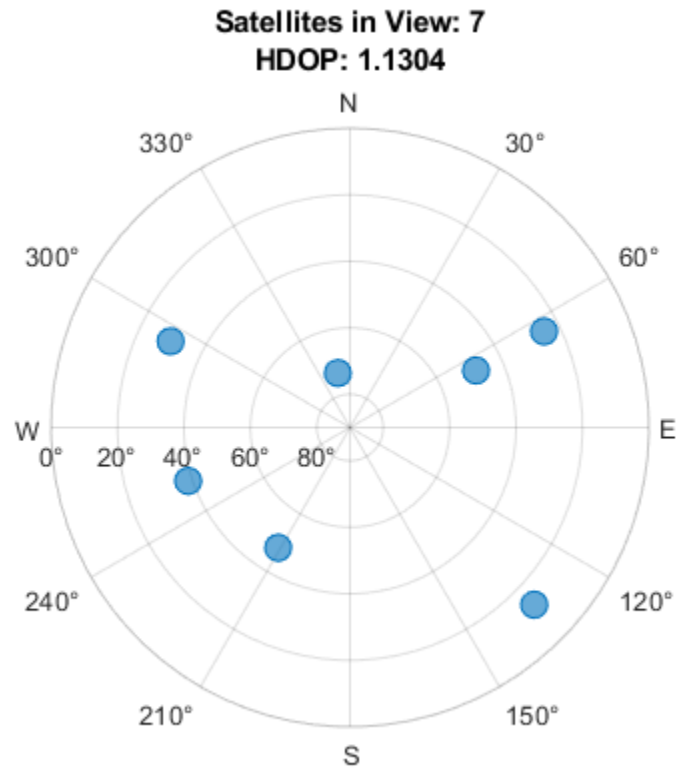
```
figure
plot(time, vertcat(status.HDOP))
title('HDOP')
ylabel('m')
xlabel('s')
```



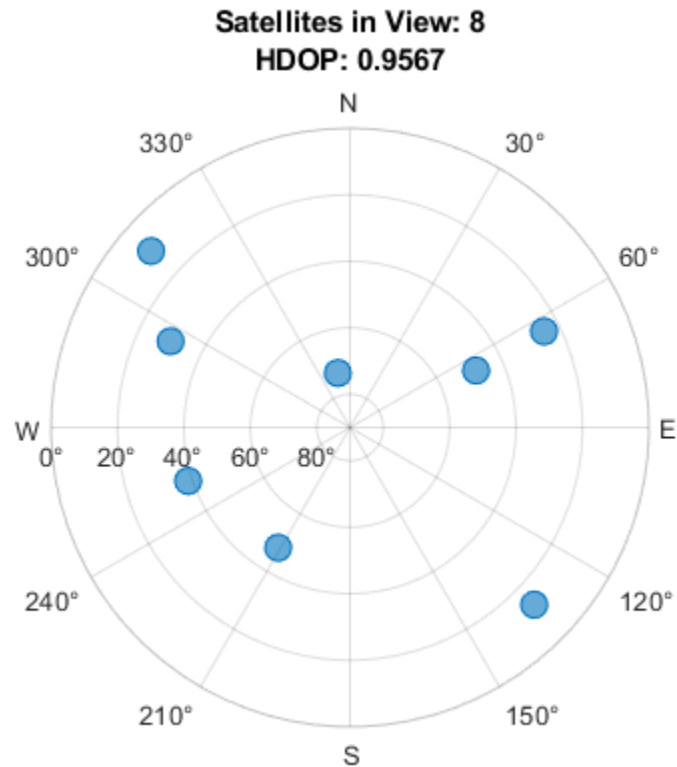
Verify that the satellite geometry has changed. Find the index where the HDOP decreased and see if that corresponds to a change in the number of satellites in view. The numSats variable increases from 7 to 8.

```
% Find expected sample index for a change in the
% number of satellites in view.
[~, satChangeIdx] = max(abs(diff(hdops)));

% Visualize the satellite geometry before the
% change in HDOP.
satAz = status(satChangeIdx).SatelliteAzimuth;
satEl = status(satChangeIdx).SatelliteElevation;
numSats = numel(satAz);
skyplot(satAz, satEl);
title(sprintf('Satellites in View: %d\nHDOP: %.4f', ...
    numSats, hdops(satChangeIdx)))
```



```
% Visualize the satellite geometry after the  
% change in HDOP.  
satAz = status(satChangeIdx+1).SatelliteAzimuth;  
satEl = status(satChangeIdx+1).SatelliteElevation;  
numSats = numel(satAz);  
skyplot(satAz, satEl);  
title(sprintf('Satellites in View: %d\nHDOP: %.4f', ...  
             numSats, hdops(satChangeIdx+1)))
```



The HDOP and VDOP values can be used as diagonal elements in measurement covariance matrices when combining GNSS receiver position estimates with other sensor measurements using a Kalman filter.

`% Convert HDOP and VDOP to a measurement covariance matrix.`

```
hdop = status(1).HDOP;
vdop = status(1).VDOP;
measCov = diag([hdop.^2/2, hdop.^2/2, vdop.^2]);
disp(measCov)
```

```
    0.6373         0         0
         0    0.6373         0
         0         0    3.6233
```

Estimate GNSS Receiver Position with Simulated Satellite Constellations

Track the position of a ground vehicle using a simulated Global Navigation Satellite System (GNSS) receiver. The satellites are simulated using the `satelliteScenario` (Satellite Communications Toolbox) object, the satellite signal processing of the receiver are simulated using the `lookangles` and `pseudoranges` functions, and the receiver position is estimated with the `receiverposition` function.

This example requires the Navigation Toolbox™.

Overview

This example focuses on the *space segment*, or satellite constellations, and the GNSS sensor equipment for a satellite system. To obtain an estimate of the GNSS receiver position, the navigation processor requires the satellite positions from the space segment and the pseudoranges from the ranging processor in the receiver.

Specify Simulation Parameters

Load the MAT-file that contains the ground-truth position and velocity of a ground vehicle travelling toward the Natick, MA campus of The MathWorks, Inc.

Specify the start, stop, and sample time of the simulation. Also, specify the mask angle, or minimum elevation angle, of the GNSS receiver.

```
% Load ground truth trajectory.
load("routeNatickMA.mat","lat","lon","pos","vel","lla0");
recPos = pos;
recVel = vel;

% Specify simulation times.
startTime = datetime(2020,10,28,8,0,0,"TimeZone","America/New_York");
simulationSteps = size(pos,1);
dt = 1;
stopTime = startTime + seconds((simulationSteps-1)*dt);

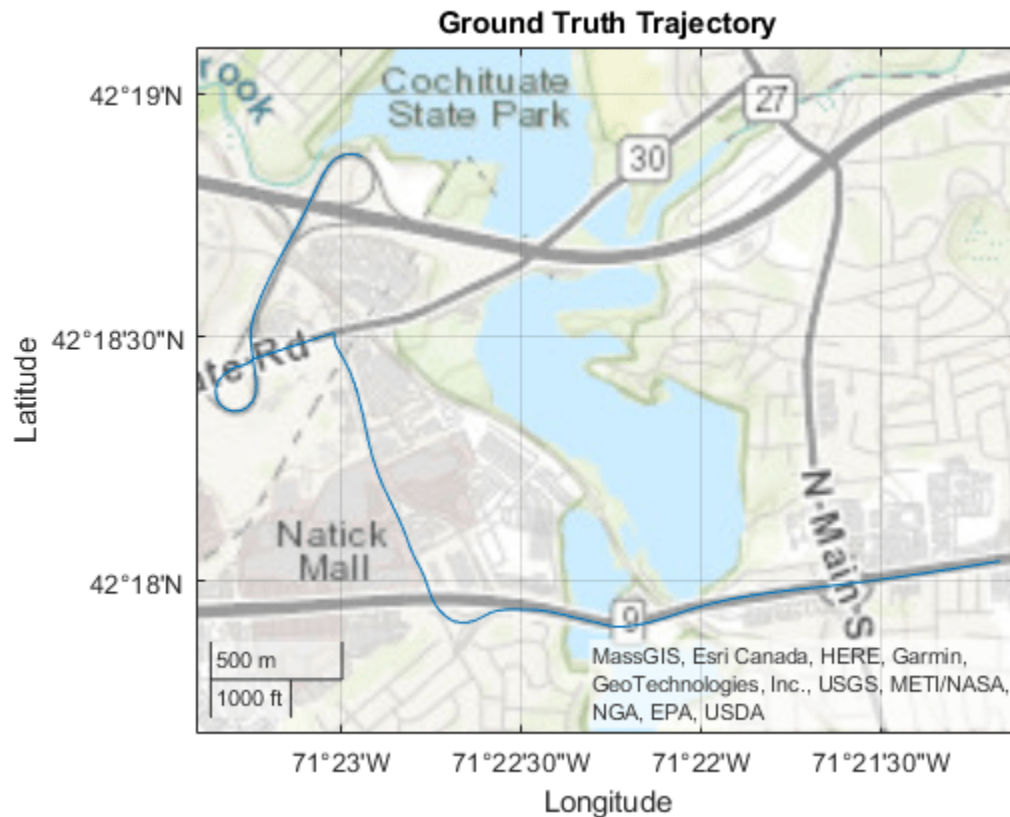
% Specify mask angle.
maskAngle = 5; % degrees

% Convert receiver position from North-East-Down (NED) to geodetic
% coordinates. Requires Navigation Toolbox(TM).
receiverLLA = ned2lla(recPos,lla0,"ellipsoid");

% Set RNG seed to allow for repeatable results.
rng("default");
```

Visualize the `geoplot` for the ground truth trajectory.

```
figure
geoplot(lat,lon)
geobasemap("topographic")
title("Ground Truth Trajectory")
```



Simulate Satellite Positions Over Time

The `satelliteScenario` object enables you to specify initial orbital parameters and visualize them using the `satelliteScenarioViewer` (Satellite Communications Toolbox) object. This example uses the `satelliteScenario` and a MAT-file with initial orbital parameters to simulate the GPS constellations over time. Alternatively, you could use the `gnssconstellation` object which simulates satellite positions using nominal orbital parameters, and only the current simulation time is needed to calculate the satellite positions.

```
% Create scenario.
sc = satelliteScenario(startTime, stopTime, dt);

load("initialOrbitalParameters.mat", "semiMajorAxis", "eccentricity", ...
     "inclination", "rightAscensionOfAscendingNode", ...
     "argumentOfPeriapsis", "trueAnomaly", "prnStr");

% Initialize satellites.
satellite(sc, semiMajorAxis, eccentricity, inclination, ...
          rightAscensionOfAscendingNode, argumentOfPeriapsis, trueAnomaly, ...
          "Name", prnStr);

% Preallocate results.
numSats = numel(sc.Satellites);
allSatPos = zeros(numSats, 3, simulationSteps);
allSatVel = zeros(numSats, 3, simulationSteps);

% Save satellite states over entire simulation.
```

```

for i = 1:numel(sc.Satellites)
    [oneSatPos, oneSatVel] = states(sc.Satellites(i), "CoordinateFrame", "ecef");
    allSatPos(i, :, :) = permute(oneSatPos, [3 1 2]);
    allSatVel(i, :, :) = permute(oneSatVel, [3 1 2]);
end

```

Calculate Pseudoranges

Use the satellite positions to calculate the pseudoranges and satellite visibilities throughout the simulation. The mask angle is used to determine the satellites that are visible to the receiver. The pseudoranges are the distances between the satellites and the GNSS receiver. The term *pseudorange* is used because this distance value is calculated by multiplying the time difference between the current receiver clock time and the timestamped satellite signal by the speed of light.

```

% Preallocate results.
allP = zeros(numSats, simulationSteps);
allPDot = zeros(numSats, simulationSteps);
allIsSatVisible = false(numSats, simulationSteps);

% Use the skyplot to visualize satellites in view.
sp = skyplot([], []);

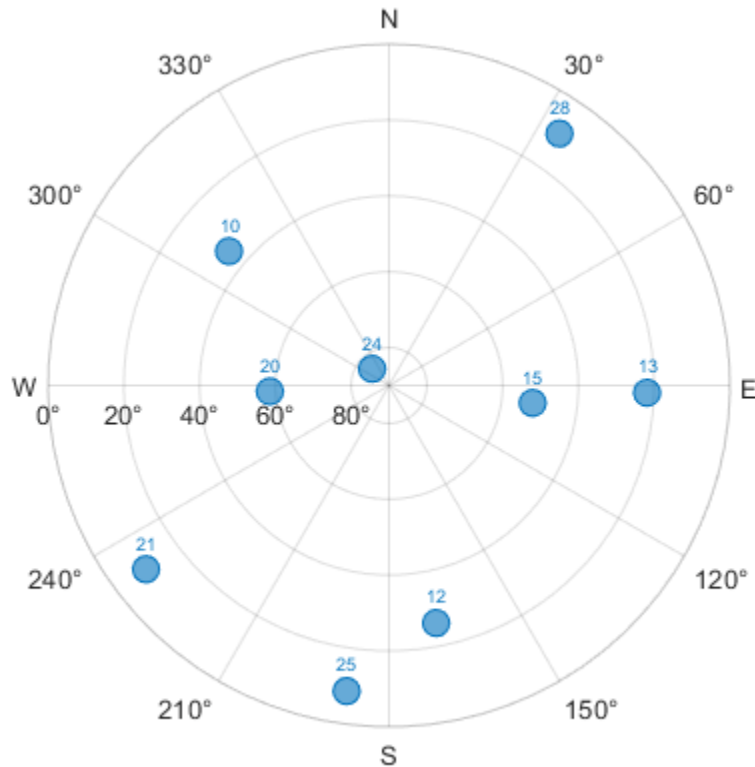
for idx = 1:simulationSteps
    satPos = allSatPos(:, :, idx);
    satVel = allSatVel(:, :, idx);

    % Calculate satellite visibilities from receiver position.
    [satAz, satEl, allIsSatVisible(:, idx)] = lookangles(receiverLLA(idx, :), satPos, maskAngle);

    % Calculate pseudoranges and pseudorange rates using satellite and
    % receiver positions and velocities.
    [allP(:, idx), allPDot(:, idx)] = pseudoranges(receiverLLA(idx, :), satPos, recVel(idx, :), satVel);

    set(sp, "AzimuthData", satAz(allIsSatVisible(:, idx)), ...
        "ElevationData", satEl(allIsSatVisible(:, idx)), ...
        "LabelData", prnStr(allIsSatVisible(:, idx)))
    drawnow limitrate
end

```



Estimate Receiver Position from Pseudoranges and Satellite Positions

Finally, use the satellite positions and pseudoranges to estimate the receiver position with the `receiverposition` function.

```
% Preallocate results.
lla = zeros(simulationSteps,3);
gnssVel = zeros(simulationSteps,3);
hdop = zeros(simulationSteps,1);
vdop = zeros(simulationSteps,1);

for idx = 1:simulationSteps
    p = allP(:,idx);
    pdot = allPDot(:,idx);
    isSatVisible = allIsSatVisible(:,idx);
    satPos = allSatPos(:,:,idx);
    satVel = allSatVel(:,:,idx);

    % Estimate receiver position and velocity using pseudoranges,
    % pseudorange rates, and satellite positions and velocities.
    [lla(idx,:),gnssVel(idx,:),hdop(idx,:),vdop(idx,:)] = receiverposition(p(isSatVisible),...
        satPos(isSatVisible,:),pdot(isSatVisible),satVel(isSatVisible,:));
end
```

Visualize Results

Plot the ground truth position and the estimated receiver position on a `geoplot`.


```

figure
geoplot(lat,lon,lla(:,1),lla(:,2))
geobasemap("topographic")
legend("Ground Truth","Estimate")

```

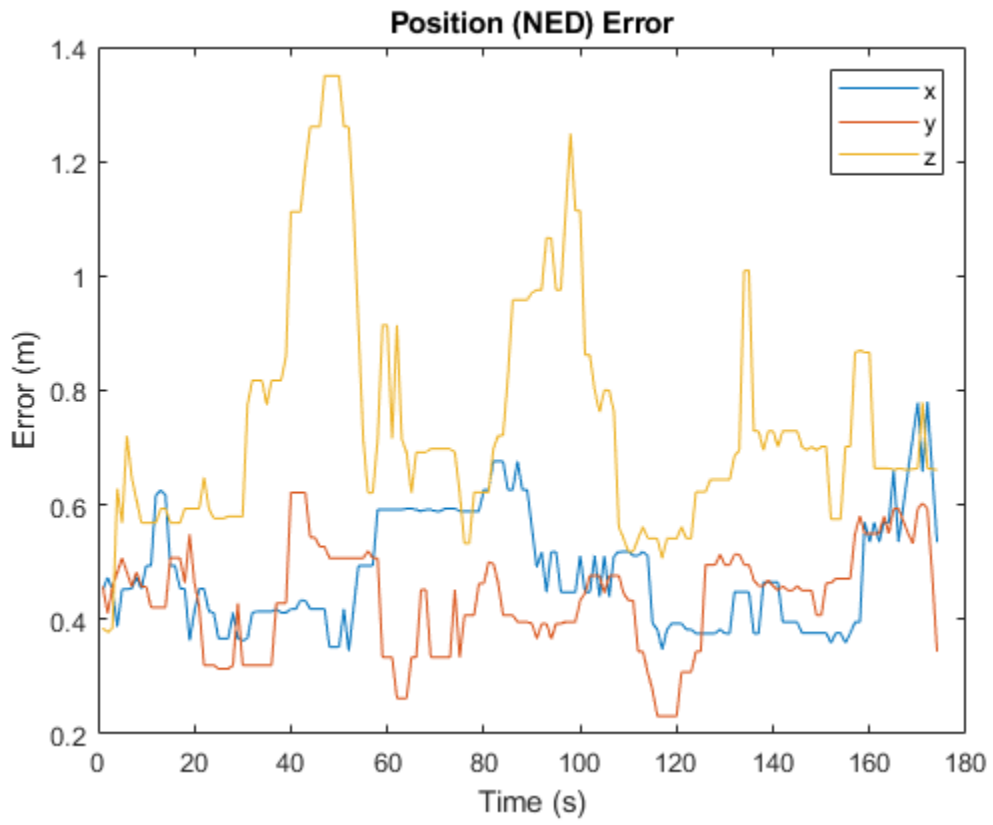


Plot the absolute error in the position estimate. The error is smoothed by a moving median to make the plot more readable. The error in the x- and y-axis is smaller because there are satellites on either side of the receiver. The error in the z-axis is larger because there are only satellites above the receiver, not below it. The error changes over time as the receiver moves and some satellites come in and out of view.

```

estPos = lla2ned(lla,lla0,"ellipsoid");
winSize = floor(size(estPos,1)/10);
figure
plot(smoothdata(abs(estPos-pos),"movmedian",winSize))
legend("x","y","z")
xlabel("Time (s)")
ylabel("Error (m)")
title("Position (NED) Error")

```



Analyze GPS Satellite Visibility

This example shows how to simulate and analyze GPS satellite visibility at specified receiver positions and times. Use live script controls to set different parameters for the satellite simulation.

Specify Simulation Parameters

Specify the start time, duration in hours, and time between samples in seconds of the simulation. Also specify the position of the receiver in geodetic coordinates and the mask angle, or minimum elevation angle, of the receiver.

```
dateString = '2021-04-08 00:00:00.000';
startTime = datetime(dateString,InputFormat="yyyy-MM-dd HH:mm:ss.SSS");
numHours = 24;
dt = 60; % s
latitude = 42.3013162; % deg
longitude = -71.3782972; % deg
altitude = 50; % m
recPos = [latitude longitude altitude]; % [deg deg m]
maskAngle = 5; % deg
```

Generate Satellite Visibilities

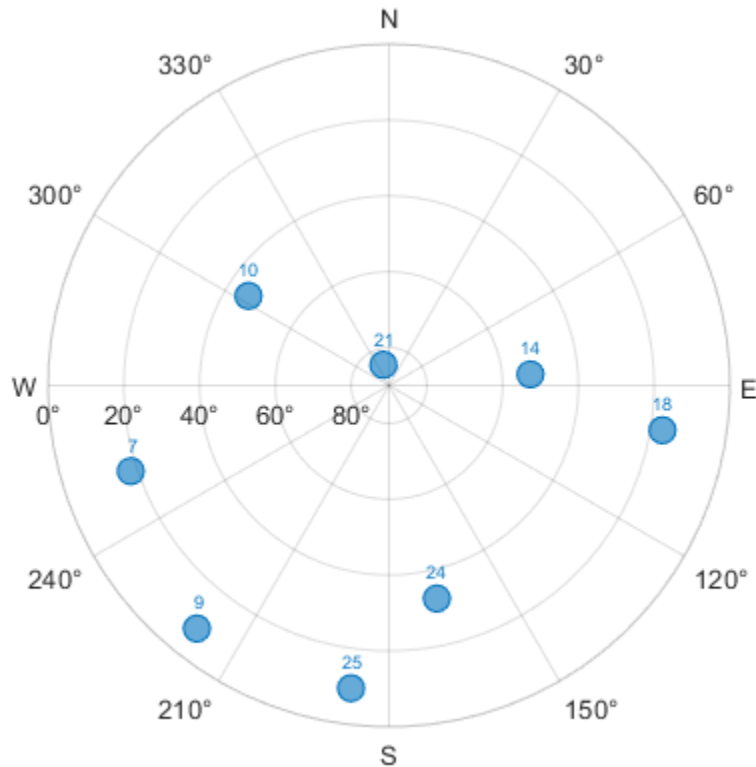
Using the parameters, generate the satellite visibilities as a matrix of logical values. Each row in the matrix corresponds to a time step, and each column corresponds to a satellite. To plot visibilities, iterate through the time vector calculating the satellite positions and look angles based on GNSS constellation simulation.

```
secondsPerHour = 3600;
timeElapsed = 0:dt:(secondsPerHour*numHours);
t = startTime + seconds(timeElapsed);

satPos = gnssconstellation(t(1));
numSats = size(satPos,1);
numSamples = numel(t);
az = zeros(numSamples,numSats);
el = zeros(numSamples,numSats);
vis = false(numSamples,numSats);
satIDs = 1:numSats;

sp = skyplot([],[]);

for ii = 1:numel(t)
    satPos = gnssconstellation(t(ii));
    [az(ii,:),el(ii,:),vis(ii,:)] = lookangles(recPos,satPos,maskAngle);
    set(sp,AzimuthData=az(ii,vis(ii,:)), ...
        ElevationData=el(ii,vis(ii,:)), ...
        LabelData=satIDs(vis(ii,:)));
    drawnow limitrate
end
```

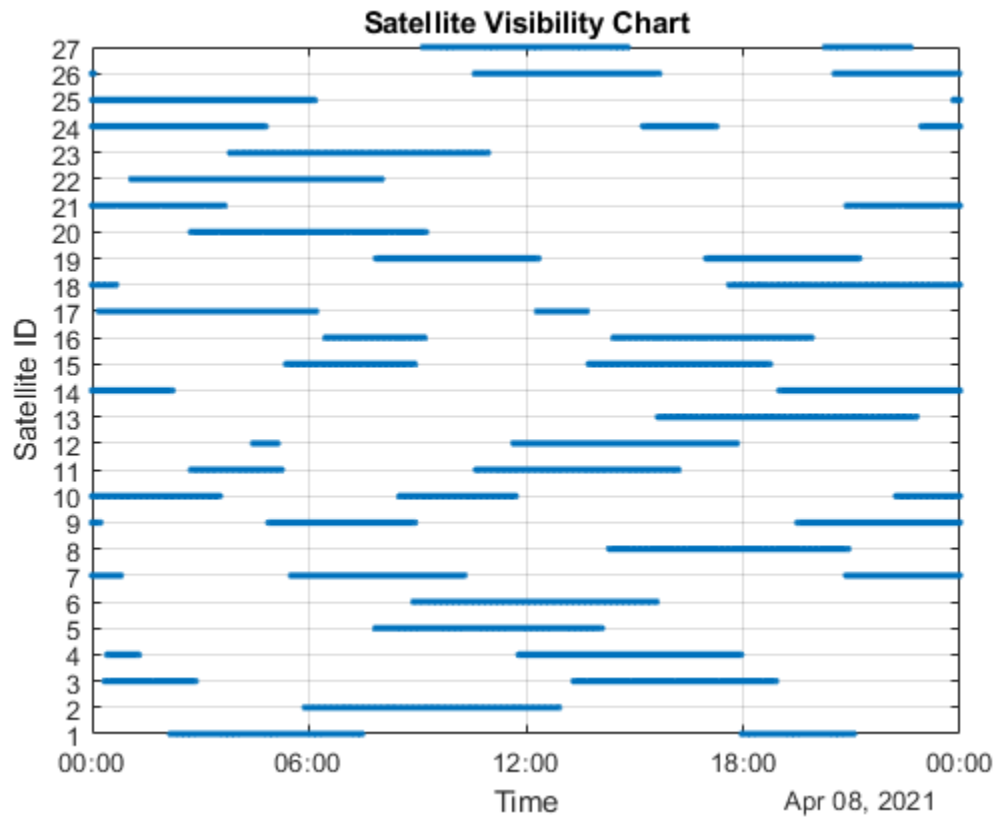


Plot Results

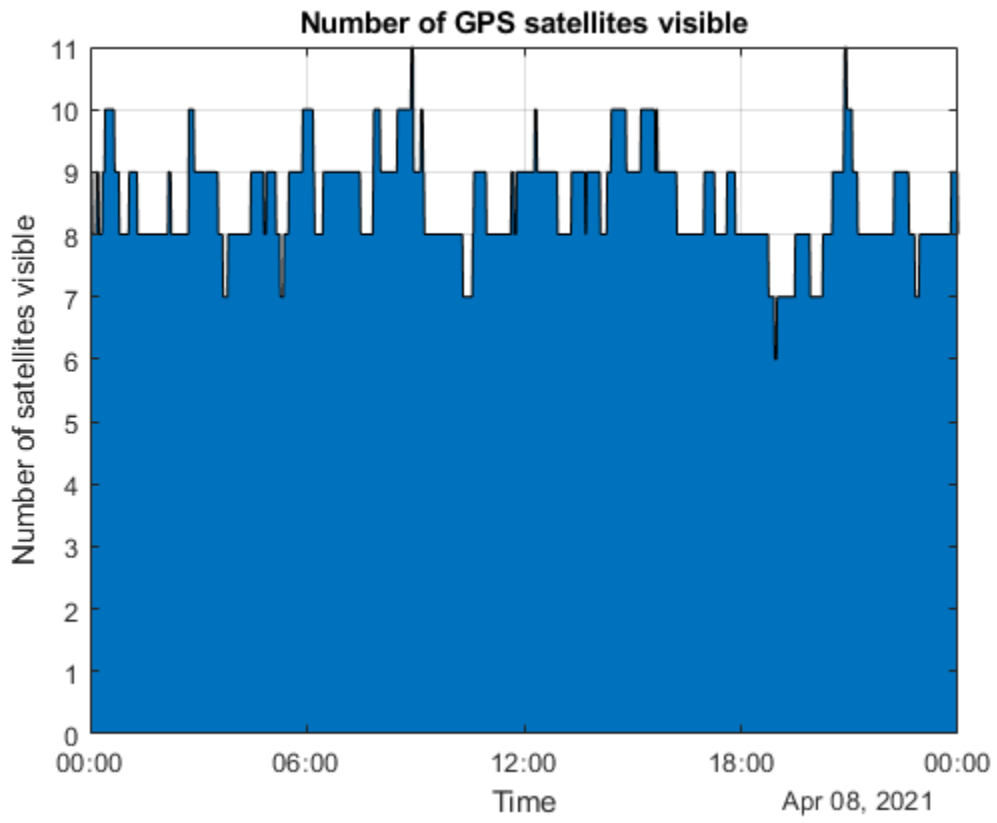
Use the logical matrix to generate a satellite visibility chart and plot the total number of visible satellites at each time step. In general, at least four satellites must be visible to compute a positioning solution.

```
visPlotData = double(vis);
visPlotData(visPlotData == false) = NaN; % Hide invisible satellites.
visPlotData = visPlotData + (0:numSats-1); % Add space to satellites to be stacked.
colors = colororder;
```

```
figure
plot(t,visPlotData,".",Color=colors(1,:))
yticks(1:numSats)
yticklabels(string(satIDs))
grid on
ylabel("Satellite ID")
xlabel("Time")
title("Satellite Visibility Chart")
axis tight
```



```
numVis = sum(vis,2);  
figure  
area(t,numVis)  
grid on  
xlabel("Time")  
ylabel("Number of satellites visible")  
title("Number of GPS satellites visible")  
axis tight
```



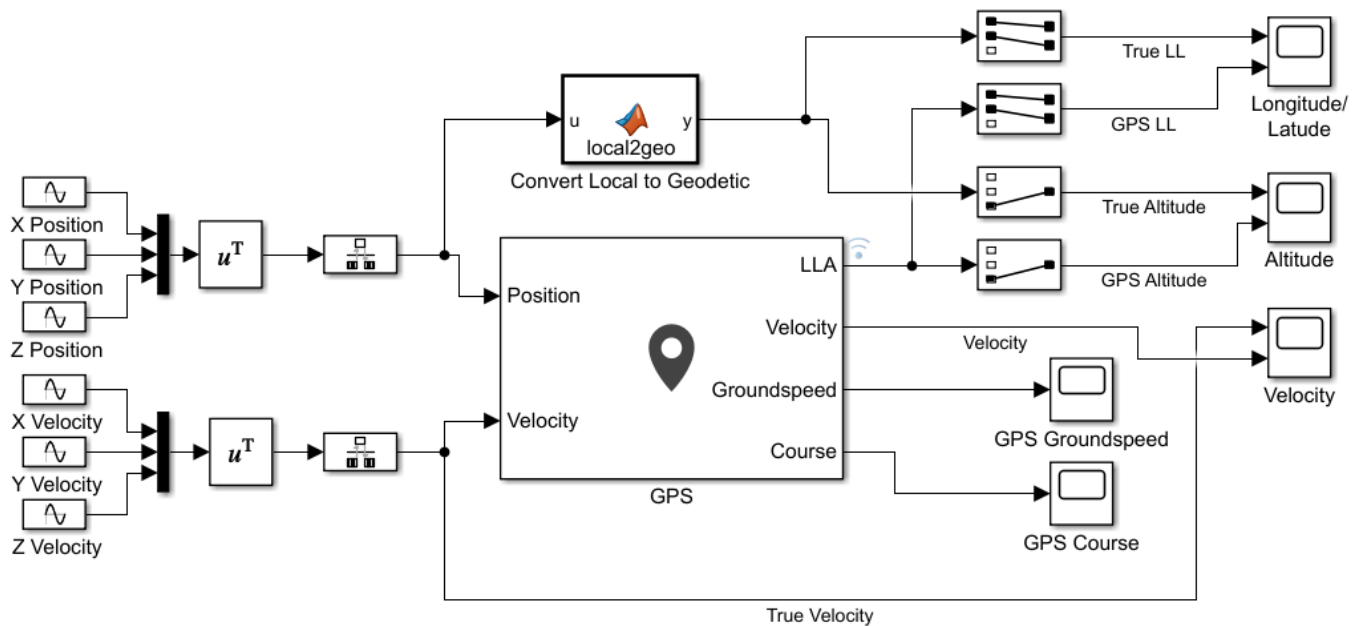
Simulate GPS Sensor Noise

This example shows how to use the GPS block to add GPS sensor noise to position and velocity inputs in Simulink®.

Examine Model

Open the Simulink model.

```
open_system("GPSNoiseModel.slx");
```



© Copyright 2021 The MathWorks, Inc.

This model generates the X, Y, Z values, for both position and velocity, as individual sine waves and combines them using Mux blocks. Because the GPS block requires discrete signals, the combined position and velocity pass through Rate Transition blocks to the inputs to the **Position** and **Velocity** ports of the GPS block. The GPS block has default parameter settings except for the **Vertical position accuracy**, which is set to 1.5 due to the scale of the position and velocity.

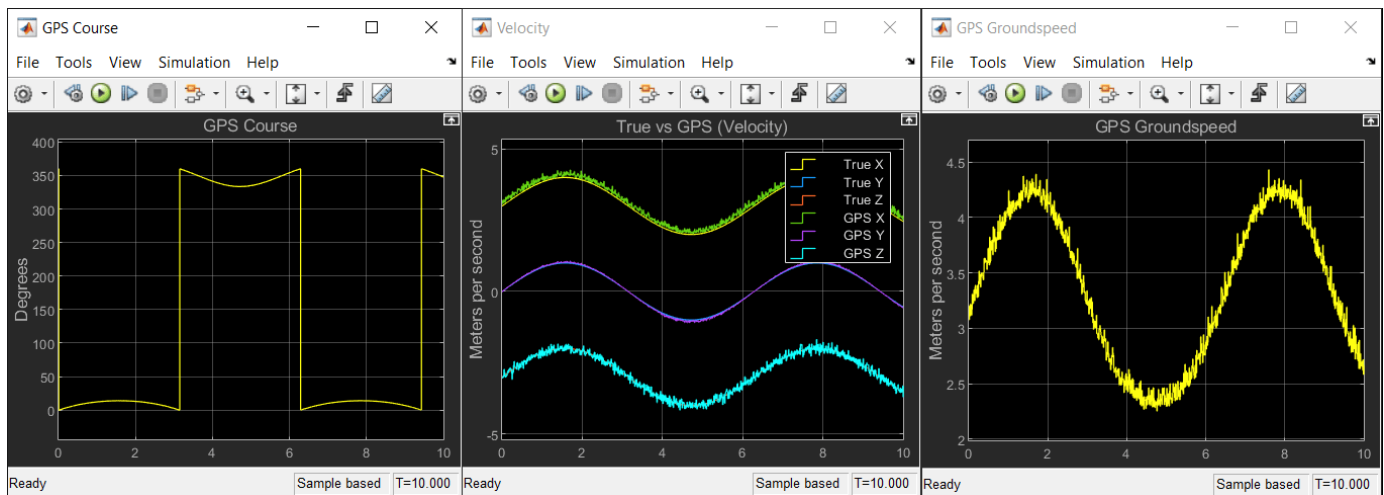
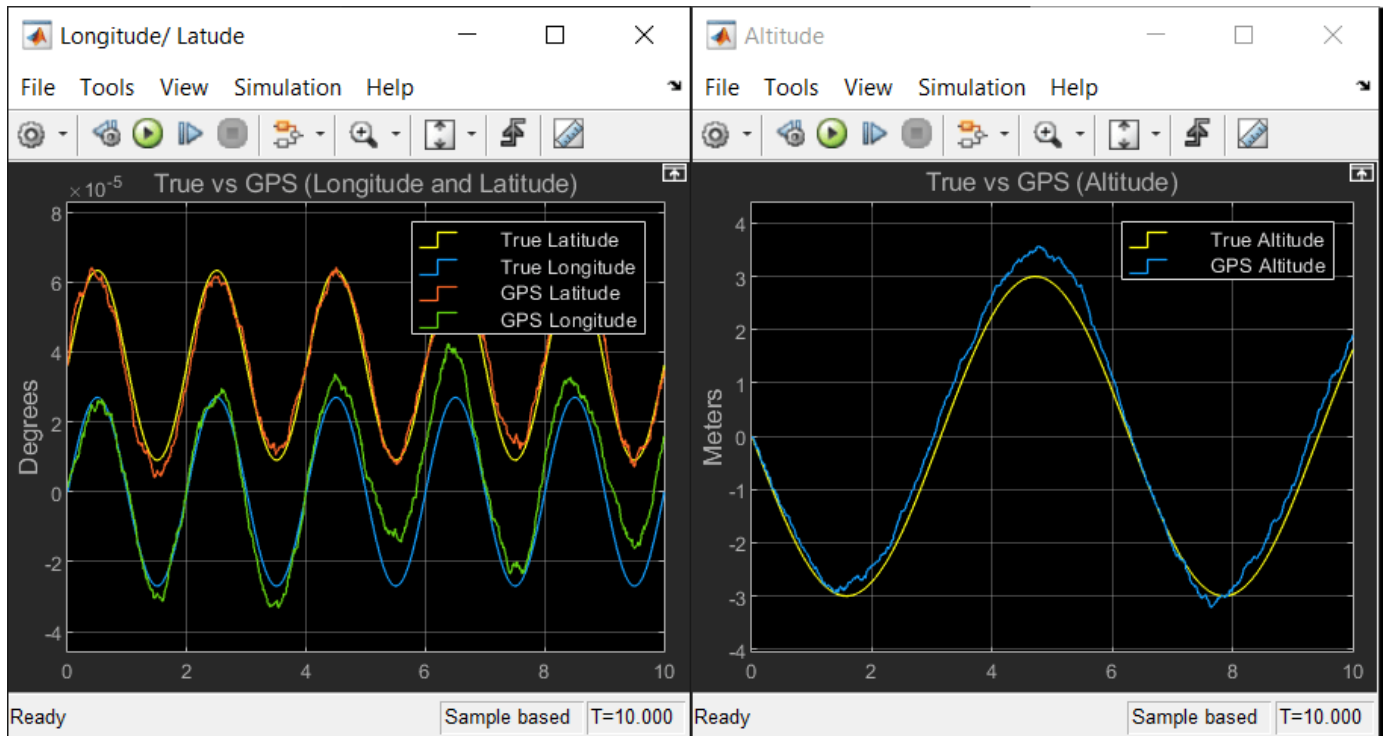
Compare the outputs of the GPS block against the true signal values using Scope blocks. To do this for position, the local position coordinates will need to be converted to LLA coordinates. Use `ned2lla` function in the MATLAB Function block to convert the NED coordinates to LLA coordinates.

A MATLAB Function block uses the `ned2lla` function to convert the local position coordinates of the true signal values to geodetic coordinates. The model then plots the outputs of the GPS block against the true signal values.

Run Model

Run the model. The output scopes show the effect of the noise from the GPS sensor on the original and velocity outputs.

1 Navigation Featured Examples



IMU Sensor Fusion with Simulink

This example shows how to generate and fuse IMU sensor data using Simulink®. You can accurately model the behavior of an accelerometer, a gyroscope, and a magnetometer and fuse their outputs to compute orientation.

Inertial Measurement Unit

An inertial measurement unit (IMU) is a group of sensors consisting of an accelerometer measuring acceleration and a gyroscope measuring angular velocity. Frequently, a magnetometer is also included to measure the Earth's magnetic field. Each of these three sensors produces a 3-axis measurement, and these three measurements constitute a 9-axis measurement.

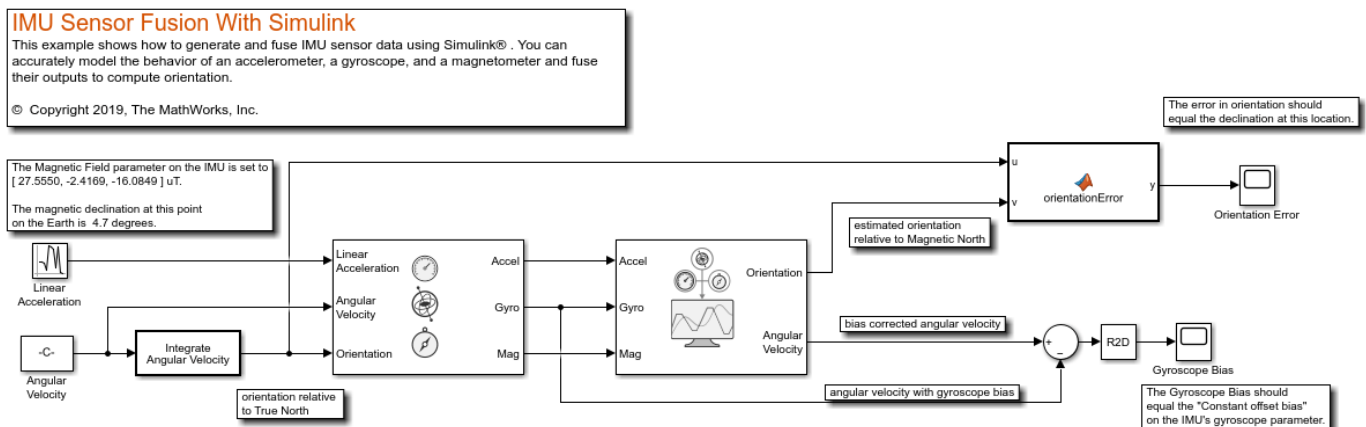
Attitude Heading and Reference System

An Attitude Heading and Reference System (AHRS) takes the 9-axis sensor readings and computes the orientation of the device. This orientation is given relative to the NED frame, where N is the Magnetic North direction. The AHRS block in Simulink accomplishes this using an indirect Kalman filter structure.

Simulink System

Open the Simulink model that fuses IMU sensor data

```
open_system('IMUFusionSimulinkModel');
```



Inputs and Configuration

The inputs to the IMU block are the device's linear acceleration, angular velocity, and the orientation relative to the navigation frame. The orientation is of the form of a quaternion (a 4-by-1 vector in Simulink) or rotation matrix (a 3-by-3 matrix in Simulink) that rotates quantities in the navigation frame to the body frame. In this model, the angular velocity is simply integrated to create an orientation input. The angular velocity is in rad/s and the linear acceleration is in m/s^2 . Because the AHRS has only one input related to translation (the accelerometer input), it cannot distinguish between gravity and linear acceleration. Therefore, the AHRS algorithm assumes that linear acceleration is a slowly varying white noise process. This is a common assumption for 9-axis fusion algorithms.

True North vs Magnetic North

Magnetic field parameter on the IMU block dialog can be set to the local magnetic field value. Magnetic field values can be found on the NOAA website or using the `wrldmagm` function in the Aerospace Toolbox™. The magnetic field values on the IMU block dialog correspond the readings of a perfect magnetometer that is orientated to True North. Therefore, the orientation input to the IMU block is relative to the NED frame, where N is the True North direction. However, the AHRS filter navigates towards Magnetic North, which is typical for this type of filter. Therefore, the orientation input to the IMU and the estimated orientation at the output of the AHRS differ by the declination angle between True North and Magnetic North.

This simulation is setup for 0° latitude and 0° longitude. The magnetic field at this location is set as [27.5550, -2.4169, -16.0849] microtesla in the IMU block. The declination at this location is about 4.7°

Simulation

Simulate the model. The IMU input orientation and the estimated output orientation of the AHRS are compared using quaternion distance. This is preferable compared to differencing the Euler angle equivalents, considering the Euler angle singularities.

```
sim('IMUFusionSimulinkModel');
```

Estimated Orientation

The difference in estimated vs true orientation should be nearly 4.7° , which is the declination at this latitude and longitude.

Gyroscope Bias

The second output of the AHRS filter is the bias-corrected gyroscope reading. In the IMU block, the gyroscope was given a bias of 0.0545 rad/s or 3.125 deg/s, which should match the steady state value in the Gyroscope Bias scope block.

Further Exercises

By varying the parameters on the IMU, you should see a corresponding change in orientation on the output of the AHRS. You can set the parameters on the IMU block to match a real IMU datasheet and tune the AHRS parameters to meet your requirements.

Automatic Tuning of the insfilterAsync Filter

The `insfilterAsync` object is a complex extended Kalman filter that estimates the device pose. However, manually tuning the filter or finding the optimal values for the noise parameters can be a challenging task. This example illustrates how to use the `tune` function to optimize the filter noise parameters.

Trajectory and Sensor Setup

To illustrate the tuning process of the `insfilterAsync` filter, use a simple random waypoint trajectory. The `imuSensor` and `gpsSensor` objects create inputs for the filter.

```
% The IMU runs at 100 Hz and the GPS runs at 1 Hz.
imurate = 100;
gpsrate = 1;
decim = imurate/gpsrate;

% Create a random waypoint trajectory.
rng(1)
Npts = 4; % number of waypoints
wpPer = 5; % time between waypoints
tstart = 0;
tend = wpPer*(Npts -1);
wp = waypointTrajectory('Waypoints',5*rand(Npts,3), ...
    'TimeOfArrival',tstart:wpPer:tend, ...
    'Orientation',[quaternion.ones; randrot(Npts-1,1)], ...
    'SampleRate', imurate);

[Position,Orientation,Velocity,Acceleration,AngularVelocity] = lookupPose(...
    wp, tstart:(1/imurate):tend);

% Set up an IMU and process the trajectory.
imu = imuSensor('SampleRate',imurate);
loadparams(imu,fullfile(matlabroot, ...
    "toolbox","shared","positioning","positioningdata","generic.json"), ...
    "GenericLowCost9Axis");
[Accelerometer, Gyroscope, Magnetometer] = imu(Acceleration, ...
    AngularVelocity, Orientation);
imuData = timetable(Accelerometer,Gyroscope,Magnetometer,'SampleRate',imurate);

% Set up a GPS sensor and process the trajectory.
gps = gpsSensor('SampleRate', gpsrate,'DecayFactor',0.5, ...
    'HorizontalPositionAccuracy',1.6,'VerticalPositionAccuracy',1.6, ...
    'VelocityAccuracy',0.1);
[GPSPosition,GPSVelocity] = gps(Position(1:decim:end,:), Velocity(1:decim:end,:));
gpsData = timetable(GPSPosition,GPSVelocity,'SampleRate',gpsrate);

% Create a timetable for the tune function.
sensorData = synchronize(imuData,gpsData);

% Create a timetable capturing the ground truth pose.
groundTruth = timetable(Position,Orientation,'SampleRate',imurate);
```

Construct the Filter

The `insfilterAsync` filter fuses data from multiple sensors operating asynchronously

```
filtUntuned = insfilterAsync;
```

Determine Filter Initial Conditions

Set the initial values for the `State` and `StateCovariance` properties based on the ground truth. Normally to obtain the initial values, you would use the first several samples of `sensorData` along with calibration routines. However, in this example the `groundTruth` is used to set the initial state for fast convergence of the filter.

```
idx = stateinfo(filtUntuned);
filtUntuned.State(idx.Orientation) = compact(Orientation(1));
filtUntuned.State(idx.AngularVelocity) = AngularVelocity(1,:);
filtUntuned.State(idx.Position) = Position(1,:);
filtUntuned.State(idx.Velocity) = Velocity(1,:);
filtUntuned.State(idx.Acceleration) = Acceleration(1,:);
filtUntuned.State(idx.AccelerometerBias) = imu.Accelerometer.ConstantBias;
filtUntuned.State(idx.GyroscopeBias) = imu.Gyroscope.ConstantBias;
filtUntuned.State(idx.GeomagneticFieldVector) = imu.MagneticField;
filtUntuned.State(idx.MagnetometerBias) = imu.Magnetometer.ConstantBias;
filtUntuned.StateCovariance = 1e-5*eye(numel(filtUntuned.State));

% Create a copy of the filtUntuned object for tuning later.
filtTuned = copy(filtUntuned);
```

Process sensorData with an Untuned Filter

Use the `tunernoise` function to create measurement noises which also need to be tuned. To illustrate the necessity for tuning, first use the filter with its default parameters.

```
mn = tunernoise('insfilterAsync');
[posUntunedEst, orientUntunedEst] = fuse(filtUntuned, sensorData, mn);
```

Tune the Filter and Process sensorData

Use the `tune` function to minimize the root mean squared (RMS) error between the `groundTruth` and state estimates.

```
cfg = tunerconfig(class(filtTuned), 'MaxIterations', 15, 'StepForward', 1.1);
tunedmn = tune(filtTuned, mn, sensorData, groundTruth, cfg);
```

Iteration	Parameter	Metric
1	AccelerometerNoise	4.5898
1	GyroscopeNoise	4.5694
1	MagnetometerNoise	4.5481
1	GPSPositionNoise	4.4737
1	GPSVelocityNoise	4.2984
1	QuaternionNoise	4.2984
1	AngularVelocityNoise	3.9668
1	PositionNoise	3.9668
1	VelocityNoise	3.9668
1	AccelerationNoise	3.9556
1	GyroscopeBiasNoise	3.9556
1	AccelerometerBiasNoise	3.9511
1	GeomagneticVectorNoise	3.9511
1	MagnetometerBiasNoise	3.9360
2	AccelerometerNoise	3.9360
2	GyroscopeNoise	3.9360
2	MagnetometerNoise	3.9064
2	GPSPositionNoise	3.9064

2	GPSVelocityNoise	3.7129
2	QuaternionNoise	3.7122
2	AngularVelocityNoise	3.0116
2	PositionNoise	3.0116
2	VelocityNoise	3.0116
2	AccelerationNoise	2.9850
2	GyroscopeBiasNoise	2.9850
2	AccelerometerBiasNoise	2.9824
2	GeomagneticVectorNoise	2.9824
2	MagnetometerBiasNoise	2.9821
3	AccelerometerNoise	2.9821
3	GyroscopeNoise	2.9613
3	MagnetometerNoise	2.9432
3	GPSPositionNoise	2.9432
3	GPSVelocityNoise	2.8373
3	QuaternionNoise	2.8369
3	AngularVelocityNoise	2.6993
3	PositionNoise	2.6993
3	VelocityNoise	2.6993
3	AccelerationNoise	2.6993
3	GyroscopeBiasNoise	2.6993
3	AccelerometerBiasNoise	2.6971
3	GeomagneticVectorNoise	2.6971
3	MagnetometerBiasNoise	2.6955
4	AccelerometerNoise	2.6941
4	GyroscopeNoise	2.6797
4	MagnetometerNoise	2.6676
4	GPSPositionNoise	2.6626
4	GPSVelocityNoise	2.5530
4	QuaternionNoise	2.5530
4	AngularVelocityNoise	2.4285
4	PositionNoise	2.4285
4	VelocityNoise	2.4285
4	AccelerationNoise	2.4232
4	GyroscopeBiasNoise	2.4232
4	AccelerometerBiasNoise	2.4217
4	GeomagneticVectorNoise	2.4217
4	MagnetometerBiasNoise	2.4023
5	AccelerometerNoise	2.4019
5	GyroscopeNoise	2.3900
5	MagnetometerNoise	2.3900
5	GPSPositionNoise	2.3887
5	GPSVelocityNoise	2.2986
5	QuaternionNoise	2.2986
5	AngularVelocityNoise	2.1945
5	PositionNoise	2.1945
5	VelocityNoise	2.1945
5	AccelerationNoise	2.1863
5	GyroscopeBiasNoise	2.1863
5	AccelerometerBiasNoise	2.1856
5	GeomagneticVectorNoise	2.1856
5	MagnetometerBiasNoise	2.1615
6	AccelerometerNoise	2.1613
6	GyroscopeNoise	2.1393
6	MagnetometerNoise	2.1199
6	GPSPositionNoise	2.1112
6	GPSVelocityNoise	2.0140
6	QuaternionNoise	2.0140

6	AngularVelocityNoise	1.9288
6	PositionNoise	1.9288
6	VelocityNoise	1.9288
6	AccelerationNoise	1.9242
6	GyroscopeBiasNoise	1.9242
6	AccelerometerBiasNoise	1.9216
6	GeomagneticVectorNoise	1.9216
6	MagnetometerBiasNoise	1.9054
7	AccelerometerNoise	1.9047
7	GyroscopeNoise	1.8887
7	MagnetometerNoise	1.8820
7	GPSPositionNoise	1.8803
7	GPSVelocityNoise	1.7713
7	QuaternionNoise	1.7712
7	AngularVelocityNoise	1.7145
7	PositionNoise	1.7145
7	VelocityNoise	1.7145
7	AccelerationNoise	1.7122
7	GyroscopeBiasNoise	1.7122
7	AccelerometerBiasNoise	1.7112
7	GeomagneticVectorNoise	1.7112
7	MagnetometerBiasNoise	1.6873
8	AccelerometerNoise	1.6853
8	GyroscopeNoise	1.6790
8	MagnetometerNoise	1.6694
8	GPSPositionNoise	1.6565
8	GPSVelocityNoise	1.5700
8	QuaternionNoise	1.5676
8	AngularVelocityNoise	1.5366
8	PositionNoise	1.5366
8	VelocityNoise	1.5366
8	AccelerationNoise	1.5366
8	GyroscopeBiasNoise	1.5366
8	AccelerometerBiasNoise	1.5353
8	GeomagneticVectorNoise	1.5337
8	MagnetometerBiasNoise	1.5166
9	AccelerometerNoise	1.5129
9	GyroscopeNoise	1.5117
9	MagnetometerNoise	1.5114
9	GPSPositionNoise	1.4953
9	GPSVelocityNoise	1.4106
9	QuaternionNoise	1.4106
9	AngularVelocityNoise	1.3742
9	PositionNoise	1.3742
9	VelocityNoise	1.3742
9	AccelerationNoise	1.3731
9	GyroscopeBiasNoise	1.3731
9	AccelerometerBiasNoise	1.3715
9	GeomagneticVectorNoise	1.3715
9	MagnetometerBiasNoise	1.3576
10	AccelerometerNoise	1.3528
10	GyroscopeNoise	1.3528
10	MagnetometerNoise	1.3510
10	GPSPositionNoise	1.3322
10	GPSVelocityNoise	1.2512
10	QuaternionNoise	1.2512
10	AngularVelocityNoise	1.2507
10	PositionNoise	1.2507

10	VelocityNoise	1.2507
10	AccelerationNoise	1.2497
10	GyroscopeBiasNoise	1.2497
10	AccelerometerBiasNoise	1.2480
10	GeomagneticVectorNoise	1.2480
10	MagnetometerBiasNoise	1.2301
11	AccelerometerNoise	1.2233
11	GyroscopeNoise	1.2222
11	MagnetometerNoise	1.2220
11	GPSPositionNoise	1.2008
11	GPSVelocityNoise	1.1043
11	QuaternionNoise	1.1043
11	AngularVelocityNoise	1.1038
11	PositionNoise	1.1038
11	VelocityNoise	1.1038
11	AccelerationNoise	1.1028
11	GyroscopeBiasNoise	1.1028
11	AccelerometerBiasNoise	1.1013
11	GeomagneticVectorNoise	1.1013
11	MagnetometerBiasNoise	1.0867
12	AccelerometerNoise	1.0782
12	GyroscopeNoise	1.0767
12	MagnetometerNoise	1.0733
12	GPSPositionNoise	1.0505
12	GPSVelocityNoise	0.9564
12	QuaternionNoise	0.9563
12	AngularVelocityNoise	0.9563
12	PositionNoise	0.9563
12	VelocityNoise	0.9563
12	AccelerationNoise	0.9550
12	GyroscopeBiasNoise	0.9550
12	AccelerometerBiasNoise	0.9534
12	GeomagneticVectorNoise	0.9534
12	MagnetometerBiasNoise	0.9402
13	AccelerometerNoise	0.9303
13	GyroscopeNoise	0.9291
13	MagnetometerNoise	0.9269
13	GPSPositionNoise	0.9036
13	GPSVelocityNoise	0.8072
13	QuaternionNoise	0.8071
13	AngularVelocityNoise	0.8071
13	PositionNoise	0.8071
13	VelocityNoise	0.8071
13	AccelerationNoise	0.8065
13	GyroscopeBiasNoise	0.8065
13	AccelerometerBiasNoise	0.8052
13	GeomagneticVectorNoise	0.8052
13	MagnetometerBiasNoise	0.7901
14	AccelerometerNoise	0.7806
14	GyroscopeNoise	0.7806
14	MagnetometerNoise	0.7768
14	GPSPositionNoise	0.7547
14	GPSVelocityNoise	0.6932
14	QuaternionNoise	0.6930
14	AngularVelocityNoise	0.6923
14	PositionNoise	0.6923
14	VelocityNoise	0.6923
14	AccelerationNoise	0.6906

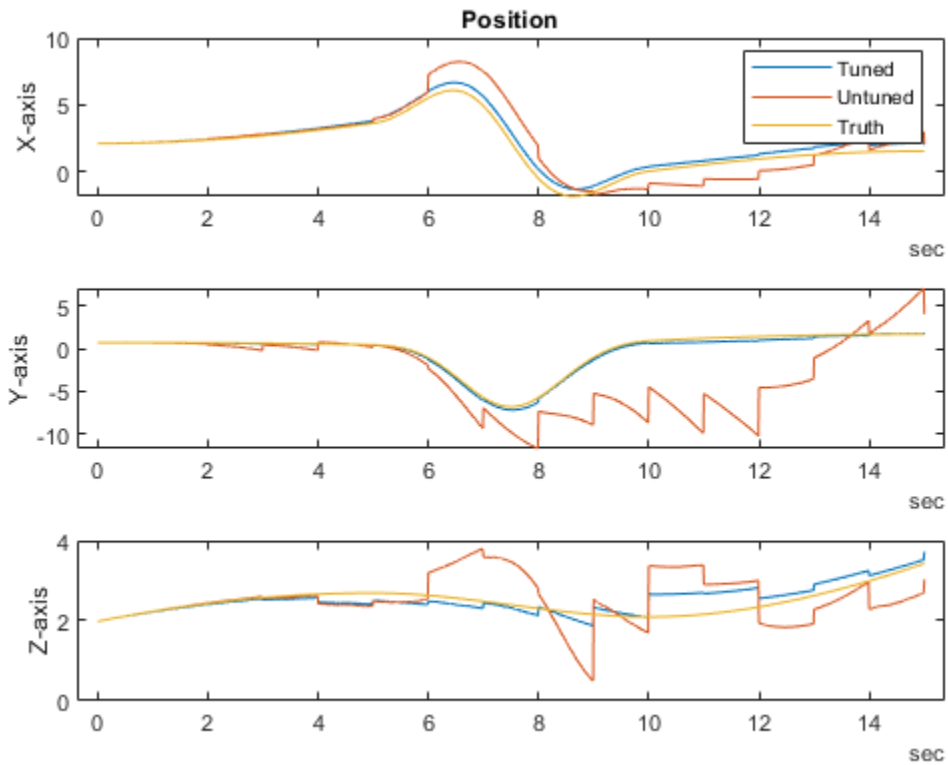
```
14 GyroscopeBiasNoise 0.6906
14 AccelerometerBiasNoise 0.6896
14 GeomagneticVectorNoise 0.6896
14 MagnetometerBiasNoise 0.6801
15 AccelerometerNoise 0.6704
15 GyroscopeNoise 0.6676
15 MagnetometerNoise 0.6653
15 GPSPositionNoise 0.6469
15 GPSVelocityNoise 0.6047
15 QuaternionNoise 0.6047
15 AngularVelocityNoise 0.6037
15 PositionNoise 0.6037
15 VelocityNoise 0.6037
15 AccelerationNoise 0.6019
15 GyroscopeBiasNoise 0.6019
15 AccelerometerBiasNoise 0.6015
15 GeomagneticVectorNoise 0.6015
15 MagnetometerBiasNoise 0.5913
```

```
[posTunedEst, orientTunedEst] = fuse(filtTuned,sensorData,tunedmn);
```

Compare Tuned vs Untuned Filter

Plot the position estimates from the tuned and untuned filters along with the ground truth positions. Then, plot the orientation error (quaternion distance) in degrees for both tuned and untuned filters. The tuned filter estimates the position and orientation better than the untuned filter.

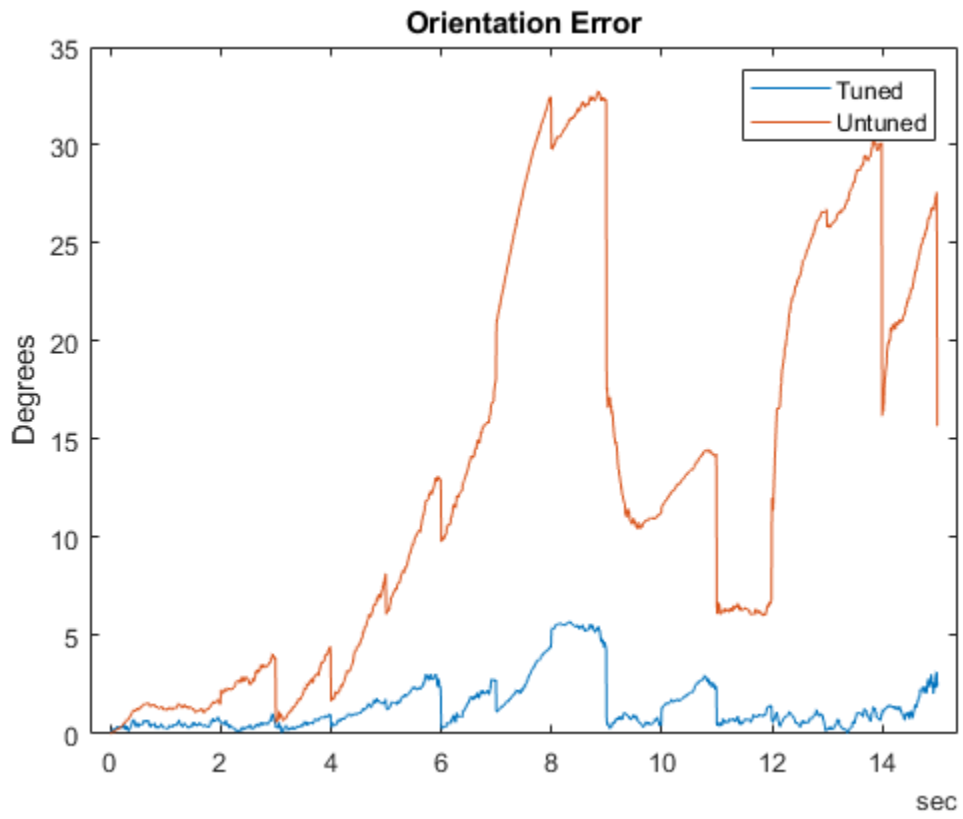
```
% Position error
figure;
t = sensorData.Time;
subplot(3,1,1);
plot(t, [posTunedEst(:,1) posUntunedEst(:,1) Position(:,1)]);
title('Position');
ylabel('X-axis');
legend('Tuned', 'Untuned', 'Truth');
subplot(3,1,2);
plot(t, [posTunedEst(:,2) posUntunedEst(:,2) Position(:,2)]);
ylabel('Y-axis');
subplot(3,1,3);
plot(t, [posTunedEst(:,3) posUntunedEst(:,3) Position(:,3)]);
ylabel('Z-axis');
```

```

% Orientation Error
figure;
plot(t, rad2deg(dist(Orientation, orientTunedEst)), ...
     t, rad2deg(dist(Orientation, orientUntunedEst)));
title('Orientation Error');
ylabel('Degrees');
legend('Tuned', 'Untuned');

```



Custom Tuning of Fusion Filters

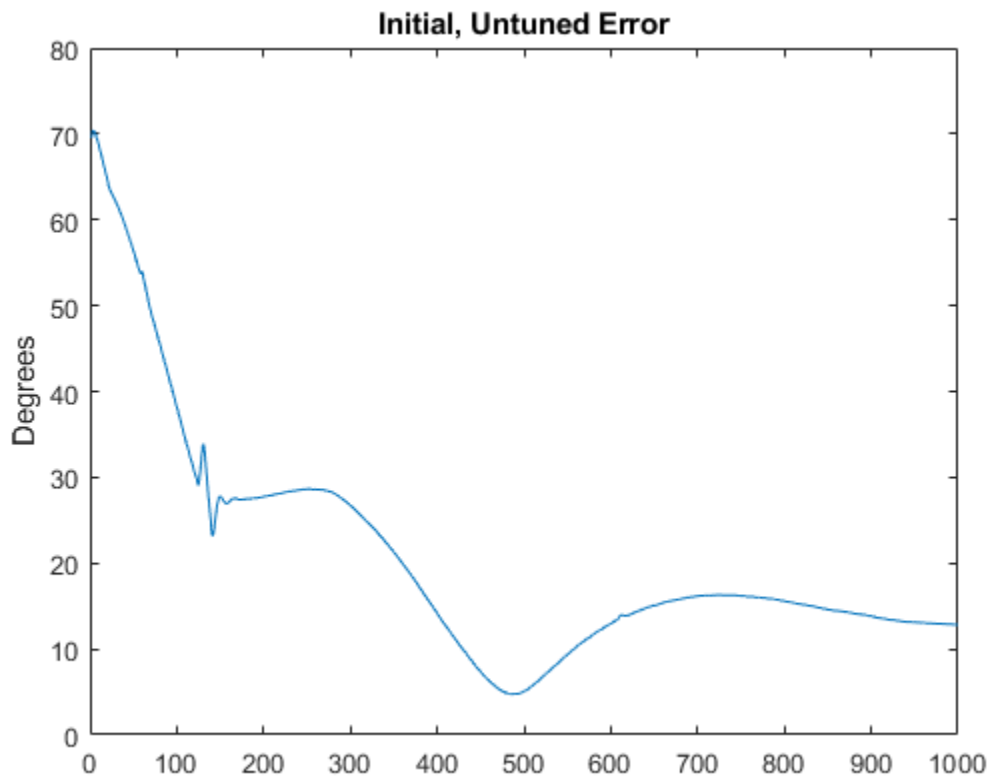
Use the `tune` function to optimize the noise parameters of several fusion filters, including the `ahrsfilter` object. This example shows how to custom a cost function for various optimization goals.

Load Sensor Data and Ground Truth

The sensor data contains sensor recordings of a UAV executing some small maneuvers. Create an `ahrsfilter` object to fuse the sensor data and estimate those maneuvers.

```
load AHRSCustomTune.mat

% Create a filter to process the data, decimating by 10.
filt = ahrsfilter('SampleRate',Fs,'DecimationFactor',10);
% Filter the sensor data and show the estimation error.
oEstInit = filt(sensorData.Accelerometer, sensorData.Gyroscope,sensorData.Magnetometer);
plotPerformance(oEstInit,groundTruth.Orientation, "Initial, Untuned Error");
```



Tune the Filter to Improve Estimation

The performance of the `ahrsfilter` without tuning noise parameters is not ideal. Use the `tune` function to improve the filter performance.

```
reset(filt);
cfg1 = tunerconfig("ahrsfilter","MaxIterations",20,"ObjectiveLimit",0.0001);
tune(filt,sensorData,groundTruth(1:10:end,:),cfg1);
```

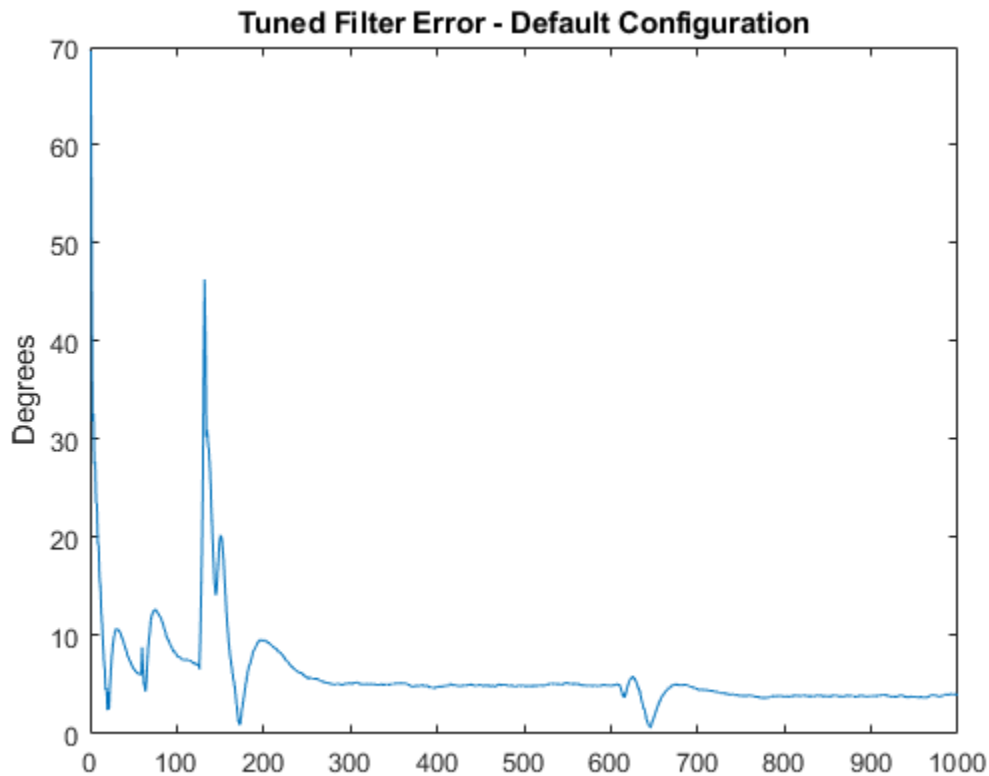
Iteration	Parameter	Metric
1	AccelerometerNoise	0.4382
1	GyroscopeNoise	0.4371
1	MagnetometerNoise	0.4370
1	GyroscopeDriftNoise	0.4370
1	LinearAccelerationNoise	0.4202
1	MagneticDisturbanceNoise	0.4188
1	LinearAccelerationDecayFactor	0.4087
1	MagneticDisturbanceDecayFactor	0.4087
2	AccelerometerNoise	0.4086
2	GyroscopeNoise	0.4066
2	MagnetometerNoise	0.4066
2	GyroscopeDriftNoise	0.4066
2	LinearAccelerationNoise	0.3937
2	MagneticDisturbanceNoise	0.3932
2	LinearAccelerationDecayFactor	0.3856
2	MagneticDisturbanceDecayFactor	0.3854
3	AccelerometerNoise	0.3853
3	GyroscopeNoise	0.3826
3	MagnetometerNoise	0.3825
3	GyroscopeDriftNoise	0.3825
3	LinearAccelerationNoise	0.3690
3	MagneticDisturbanceNoise	0.3676
3	LinearAccelerationDecayFactor	0.3613
3	MagneticDisturbanceDecayFactor	0.3611
4	AccelerometerNoise	0.3610
4	GyroscopeNoise	0.3577
4	MagnetometerNoise	0.3576
4	GyroscopeDriftNoise	0.3576
4	LinearAccelerationNoise	0.3431
4	MagneticDisturbanceNoise	0.3414
4	LinearAccelerationDecayFactor	0.3364
4	MagneticDisturbanceDecayFactor	0.3363
5	AccelerometerNoise	0.3362
5	GyroscopeNoise	0.3328
5	MagnetometerNoise	0.3326
5	GyroscopeDriftNoise	0.3326
5	LinearAccelerationNoise	0.3190
5	MagneticDisturbanceNoise	0.3183
5	LinearAccelerationDecayFactor	0.3152
5	MagneticDisturbanceDecayFactor	0.3150
6	AccelerometerNoise	0.3149
6	GyroscopeNoise	0.3121
6	MagnetometerNoise	0.3119
6	GyroscopeDriftNoise	0.3119
6	LinearAccelerationNoise	0.3040
6	MagneticDisturbanceNoise	0.3035
6	LinearAccelerationDecayFactor	0.3024
6	MagneticDisturbanceDecayFactor	0.3022
7	AccelerometerNoise	0.3022
7	GyroscopeNoise	0.2990
7	MagnetometerNoise	0.2989
7	GyroscopeDriftNoise	0.2989
7	LinearAccelerationNoise	0.2970
7	MagneticDisturbanceNoise	0.2955
7	LinearAccelerationDecayFactor	0.2952
7	MagneticDisturbanceDecayFactor	0.2948

8	AccelerometerNoise	0.2948
8	GyroscopeNoise	0.2903
8	MagnetometerNoise	0.2902
8	GyroscopeDriftNoise	0.2902
8	LinearAccelerationNoise	0.2883
8	MagneticDisturbanceNoise	0.2860
8	LinearAccelerationDecayFactor	0.2856
8	MagneticDisturbanceDecayFactor	0.2851
9	AccelerometerNoise	0.2851
9	GyroscopeNoise	0.2778
9	MagnetometerNoise	0.2777
9	GyroscopeDriftNoise	0.2777
9	LinearAccelerationNoise	0.2709
9	MagneticDisturbanceNoise	0.2698
9	LinearAccelerationDecayFactor	0.2690
9	MagneticDisturbanceDecayFactor	0.2689
10	AccelerometerNoise	0.2689
10	GyroscopeNoise	0.2593
10	MagnetometerNoise	0.2593
10	GyroscopeDriftNoise	0.2593
10	LinearAccelerationNoise	0.2492
10	MagneticDisturbanceNoise	0.2490
10	LinearAccelerationDecayFactor	0.2482
10	MagneticDisturbanceDecayFactor	0.2482
11	AccelerometerNoise	0.2481
11	GyroscopeNoise	0.2370
11	MagnetometerNoise	0.2369
11	GyroscopeDriftNoise	0.2369
11	LinearAccelerationNoise	0.2240
11	MagneticDisturbanceNoise	0.2237
11	LinearAccelerationDecayFactor	0.2230
11	MagneticDisturbanceDecayFactor	0.2228
12	AccelerometerNoise	0.2227
12	GyroscopeNoise	0.2117
12	MagnetometerNoise	0.2117
12	GyroscopeDriftNoise	0.2117
12	LinearAccelerationNoise	0.1984
12	MagneticDisturbanceNoise	0.1979
12	LinearAccelerationDecayFactor	0.1974
12	MagneticDisturbanceDecayFactor	0.1974
13	AccelerometerNoise	0.1973
13	GyroscopeNoise	0.1878
13	MagnetometerNoise	0.1878
13	GyroscopeDriftNoise	0.1878
13	LinearAccelerationNoise	0.1766
13	MagneticDisturbanceNoise	0.1763
13	LinearAccelerationDecayFactor	0.1761
13	MagneticDisturbanceDecayFactor	0.1761
14	AccelerometerNoise	0.1760
14	GyroscopeNoise	0.1686
14	MagnetometerNoise	0.1685
14	GyroscopeDriftNoise	0.1685
14	LinearAccelerationNoise	0.1601
14	MagneticDisturbanceNoise	0.1599
14	LinearAccelerationDecayFactor	0.1597
14	MagneticDisturbanceDecayFactor	0.1597
15	AccelerometerNoise	0.1596
15	GyroscopeNoise	0.1536

15	MagnetometerNoise	0.1536
15	GyroscopeDriftNoise	0.1536
15	LinearAccelerationNoise	0.1472
15	MagneticDisturbanceNoise	0.1469
15	LinearAccelerationDecayFactor	0.1469
15	MagneticDisturbanceDecayFactor	0.1469
16	AccelerometerNoise	0.1468
16	GyroscopeNoise	0.1422
16	MagnetometerNoise	0.1422
16	GyroscopeDriftNoise	0.1422
16	LinearAccelerationNoise	0.1380
16	MagneticDisturbanceNoise	0.1378
16	LinearAccelerationDecayFactor	0.1377
16	MagneticDisturbanceDecayFactor	0.1377
17	AccelerometerNoise	0.1377
17	GyroscopeNoise	0.1352
17	MagnetometerNoise	0.1351
17	GyroscopeDriftNoise	0.1351
17	LinearAccelerationNoise	0.1351
17	MagneticDisturbanceNoise	0.1351
17	LinearAccelerationDecayFactor	0.1351
17	MagneticDisturbanceDecayFactor	0.1351
18	AccelerometerNoise	0.1351
18	GyroscopeNoise	0.1351
18	MagnetometerNoise	0.1351
18	GyroscopeDriftNoise	0.1351
18	LinearAccelerationNoise	0.1351
18	MagneticDisturbanceNoise	0.1351
18	LinearAccelerationDecayFactor	0.1350
18	MagneticDisturbanceDecayFactor	0.1350
19	AccelerometerNoise	0.1350
19	GyroscopeNoise	0.1348
19	MagnetometerNoise	0.1344
19	GyroscopeDriftNoise	0.1344
19	LinearAccelerationNoise	0.1344
19	MagneticDisturbanceNoise	0.1344
19	LinearAccelerationDecayFactor	0.1344
19	MagneticDisturbanceDecayFactor	0.1344
20	AccelerometerNoise	0.1344
20	GyroscopeNoise	0.1344
20	MagnetometerNoise	0.1344
20	GyroscopeDriftNoise	0.1344
20	LinearAccelerationNoise	0.1344
20	MagneticDisturbanceNoise	0.1344
20	LinearAccelerationDecayFactor	0.1344
20	MagneticDisturbanceDecayFactor	0.1344

Filter the sensor data using the tuned filter and show the orientation error.

```
oEstTuned = filt(sensorData.Accelerometer, sensorData.Gyroscope, sensorData.Magnetometer);  
plotPerformance(oEstTuned, groundTruth.Orientation, "Tuned Filter Error - Default Configuration");
```



Use the CustomCostFcn and MATLAB Coder (R) to Accelerate and Optimize Tuning

The performance of the filter is improved after tuning but the tuning process can often take a long time. The `tunerconfig` object allows for a custom cost function to optimize this process. You can also use MATLAB Coder to create a mex function to accelerate the tuning speed. The custom cost function must have a signature `cost = fcn(params, sensorData, groundTruth)`, where `cost` is a scalar real number, `params` is a struct of noise parameters to be optimized, `sensorData` is a table of sensor data, and `groundTruth` is a table ground truth data.

From the last section, the `ahrsfilter` did not estimate the orientation very well during some of the maneuvers. Instead of using the default root-mean-squared error, the custom cost function uses higher order terms to more severely penalize outliers.

Display the details of the custom cost function. The function is attached as an m-file.

```
type customFcn.m
```

```
function c = customFcn(params, sensorData, groundTruth)
% Custom Cost function for optimizing the ahrsfilter

% Set any nontunable parameters in the constructor
decim = 10;
h = ahrsfilter('SampleRate', 200, 'DecimationFactor', decim);

% Parameterize the filter instance with the current-best parameters from
% the params struct.
h.AccelerometerNoise = params.AccelerometerNoise;
```

```

h.GyroscopeNoise = params.GyroscopeNoise;
h.MagnetometerNoise = params.MagnetometerNoise;
h.GyroscopeDriftNoise = params.GyroscopeDriftNoise;
h.LinearAccelerationNoise = params.LinearAccelerationNoise;
h.MagneticDisturbanceNoise = params.MagneticDisturbanceNoise;
h.LinearAccelerationDecayFactor = params.LinearAccelerationDecayFactor;
h.MagneticDisturbanceDecayFactor = params.MagneticDisturbanceDecayFactor;
h.ExpectedMagneticFieldStrength = params.ExpectedMagneticFieldStrength;

```

```

% Fuse sensor data
qest = h(sensorData.Accelerometer, sensorData.Gyroscope, ...
        sensorData.Magnetometer);

```

```

% Compute the orientation error
d = dist(qest, groundTruth.Orientation(1:decim:end));

```

```

% Penalize outliers heavily by using the 6th power.
c = sqrt(sqrt( mean(d(10:end,:).^6) ));

```

To create a mex file, you need exemplar input arguments. Create a parameters exemplar and copy the properties from `filt`.

```

p = {'AccelerometerNoise', 'DecimationFactor', 'ExpectedMagneticFieldStrength', ...
    'GyroscopeDriftNoise', 'GyroscopeNoise', 'InitialProcessNoise', 'LinearAccelerationDecayFactor', ...
    'LinearAccelerationNoise', 'MagneticDisturbanceDecayFactor', 'MagneticDisturbanceNoise', ...
    'MagnetometerNoise', 'OrientationFormat', 'SampleRate'}

```

```

p = 1x13 cell
    {'AccelerometerNoise'}    {'DecimationFactor'}    {'ExpectedMagneticFieldStrength'}    {'GyroscopeDriftNoise'}

```

```

for idx=1:numel(p)
    paramEx.(p{idx}) = filt.(p{idx});
end
disp(paramEx);

```

```

AccelerometerNoise: 5.4972e-07
DecimationFactor: 10
ExpectedMagneticFieldStrength: 50
GyroscopeDriftNoise: 4.0927e-11
GyroscopeNoise: 0.0041
InitialProcessNoise: [12x12 double]
LinearAccelerationDecayFactor: 0.0050
LinearAccelerationNoise: 1.4370e-04
MagneticDisturbanceDecayFactor: 0.9872
MagneticDisturbanceNoise: 0.0360
MagnetometerNoise: 0.1278
OrientationFormat: 'quaternion'
SampleRate: 200

```

Generate code.

```

codegen customFcn.m -args {paramEx sensorData, groundTruth}

```

Code generation successful.

Use the mex function to tune quickly.

```

cfg = tunerconfig("ahrsfilter", "Cost", "Custom", ...
    "CustomCostFcn", @customFcn_mex, "MaxIterations", 20, "ObjectiveLimit", 0.0001);

```



```
reset(filt);
tune(filt, sensorData, groundTruth, cfg);
```

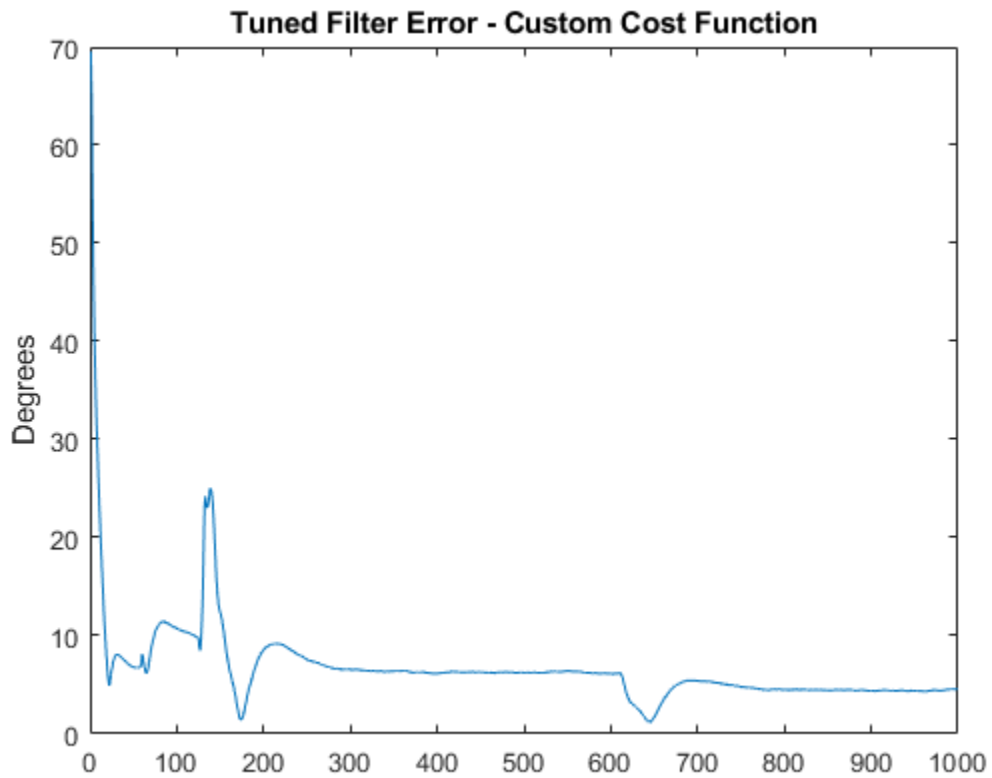
Iteration	Parameter	Metric
1	AccelerometerNoise	0.1581
1	GyroscopeNoise	0.1544
1	MagnetometerNoise	0.1544
1	GyroscopeDriftNoise	0.1544
1	LinearAccelerationNoise	0.1504
1	MagneticDisturbanceNoise	0.1498
1	LinearAccelerationDecayFactor	0.1497
1	MagneticDisturbanceDecayFactor	0.1474
2	AccelerometerNoise	0.1474
2	GyroscopeNoise	0.1437
2	MagnetometerNoise	0.1436
2	GyroscopeDriftNoise	0.1436
2	LinearAccelerationNoise	0.1395
2	MagneticDisturbanceNoise	0.1387
2	LinearAccelerationDecayFactor	0.1387
2	MagneticDisturbanceDecayFactor	0.1367
3	AccelerometerNoise	0.1367
3	GyroscopeNoise	0.1332
3	MagnetometerNoise	0.1332
3	GyroscopeDriftNoise	0.1332
3	LinearAccelerationNoise	0.1293
3	MagneticDisturbanceNoise	0.1284
3	LinearAccelerationDecayFactor	0.1284
3	MagneticDisturbanceDecayFactor	0.1271
4	AccelerometerNoise	0.1270
4	GyroscopeNoise	0.1243
4	MagnetometerNoise	0.1242
4	GyroscopeDriftNoise	0.1242
4	LinearAccelerationNoise	0.1209
4	MagneticDisturbanceNoise	0.1201
4	LinearAccelerationDecayFactor	0.1201
4	MagneticDisturbanceDecayFactor	0.1193
5	AccelerometerNoise	0.1193
5	GyroscopeNoise	0.1180
5	MagnetometerNoise	0.1178
5	GyroscopeDriftNoise	0.1178
5	LinearAccelerationNoise	0.1158
5	MagneticDisturbanceNoise	0.1152
5	LinearAccelerationDecayFactor	0.1152
5	MagneticDisturbanceDecayFactor	0.1147
6	AccelerometerNoise	0.1147
6	GyroscopeNoise	0.1147
6	MagnetometerNoise	0.1143
6	GyroscopeDriftNoise	0.1143
6	LinearAccelerationNoise	0.1132
6	MagneticDisturbanceNoise	0.1123
6	LinearAccelerationDecayFactor	0.1123
6	MagneticDisturbanceDecayFactor	0.1118
7	AccelerometerNoise	0.1118
7	GyroscopeNoise	0.1113
7	MagnetometerNoise	0.1108
7	GyroscopeDriftNoise	0.1108
7	LinearAccelerationNoise	0.1100

7	MagneticDisturbanceNoise	0.1093
7	LinearAccelerationDecayFactor	0.1093
7	MagneticDisturbanceDecayFactor	0.1093
8	AccelerometerNoise	0.1093
8	GyroscopeNoise	0.1088
8	MagnetometerNoise	0.1084
8	GyroscopeDriftNoise	0.1084
8	LinearAccelerationNoise	0.1084
8	MagneticDisturbanceNoise	0.1084
8	LinearAccelerationDecayFactor	0.1084
8	MagneticDisturbanceDecayFactor	0.1084
9	AccelerometerNoise	0.1084
9	GyroscopeNoise	0.1076
9	MagnetometerNoise	0.1072
9	GyroscopeDriftNoise	0.1072
9	LinearAccelerationNoise	0.1072
9	MagneticDisturbanceNoise	0.1072
9	LinearAccelerationDecayFactor	0.1072
9	MagneticDisturbanceDecayFactor	0.1071
10	AccelerometerNoise	0.1071
10	GyroscopeNoise	0.1068
10	MagnetometerNoise	0.1065
10	GyroscopeDriftNoise	0.1065
10	LinearAccelerationNoise	0.1062
10	MagneticDisturbanceNoise	0.1060
10	LinearAccelerationDecayFactor	0.1060
10	MagneticDisturbanceDecayFactor	0.1059
11	AccelerometerNoise	0.1059
11	GyroscopeNoise	0.1057
11	MagnetometerNoise	0.1055
11	GyroscopeDriftNoise	0.1055
11	LinearAccelerationNoise	0.1049
11	MagneticDisturbanceNoise	0.1048
11	LinearAccelerationDecayFactor	0.1048
11	MagneticDisturbanceDecayFactor	0.1048
12	AccelerometerNoise	0.1048
12	GyroscopeNoise	0.1047
12	MagnetometerNoise	0.1045
12	GyroscopeDriftNoise	0.1045
12	LinearAccelerationNoise	0.1038
12	MagneticDisturbanceNoise	0.1036
12	LinearAccelerationDecayFactor	0.1036
12	MagneticDisturbanceDecayFactor	0.1035
13	AccelerometerNoise	0.1035
13	GyroscopeNoise	0.1035
13	MagnetometerNoise	0.1033
13	GyroscopeDriftNoise	0.1033
13	LinearAccelerationNoise	0.1029
13	MagneticDisturbanceNoise	0.1027
13	LinearAccelerationDecayFactor	0.1027
13	MagneticDisturbanceDecayFactor	0.1027
14	AccelerometerNoise	0.1027
14	GyroscopeNoise	0.1024
14	MagnetometerNoise	0.1021
14	GyroscopeDriftNoise	0.1021
14	LinearAccelerationNoise	0.1019
14	MagneticDisturbanceNoise	0.1018
14	LinearAccelerationDecayFactor	0.1018

14	MagneticDisturbanceDecayFactor	0.1018
15	AccelerometerNoise	0.1018
15	GyroscopeNoise	0.1014
15	MagnetometerNoise	0.1012
15	GyroscopeDriftNoise	0.1012
15	LinearAccelerationNoise	0.1012
15	MagneticDisturbanceNoise	0.1012
15	LinearAccelerationDecayFactor	0.1011
15	MagneticDisturbanceDecayFactor	0.1011
16	AccelerometerNoise	0.1011
16	GyroscopeNoise	0.1008
16	MagnetometerNoise	0.1008
16	GyroscopeDriftNoise	0.1008
16	LinearAccelerationNoise	0.1006
16	MagneticDisturbanceNoise	0.1005
16	LinearAccelerationDecayFactor	0.1004
16	MagneticDisturbanceDecayFactor	0.1004
17	AccelerometerNoise	0.1004
17	GyroscopeNoise	0.1001
17	MagnetometerNoise	0.1001
17	GyroscopeDriftNoise	0.1001
17	LinearAccelerationNoise	0.0998
17	MagneticDisturbanceNoise	0.0998
17	LinearAccelerationDecayFactor	0.0996
17	MagneticDisturbanceDecayFactor	0.0995
18	AccelerometerNoise	0.0995
18	GyroscopeNoise	0.0992
18	MagnetometerNoise	0.0992
18	GyroscopeDriftNoise	0.0992
18	LinearAccelerationNoise	0.0990
18	MagneticDisturbanceNoise	0.0989
18	LinearAccelerationDecayFactor	0.0987
18	MagneticDisturbanceDecayFactor	0.0986
19	AccelerometerNoise	0.0986
19	GyroscopeNoise	0.0980
19	MagnetometerNoise	0.0980
19	GyroscopeDriftNoise	0.0980
19	LinearAccelerationNoise	0.0980
19	MagneticDisturbanceNoise	0.0980
19	LinearAccelerationDecayFactor	0.0978
19	MagneticDisturbanceDecayFactor	0.0975
20	AccelerometerNoise	0.0975
20	GyroscopeNoise	0.0965
20	MagnetometerNoise	0.0965
20	GyroscopeDriftNoise	0.0965
20	LinearAccelerationNoise	0.0964
20	MagneticDisturbanceNoise	0.0964
20	LinearAccelerationDecayFactor	0.0964
20	MagneticDisturbanceDecayFactor	0.0964

Filter the data and show the orientation error.

```
oEst = filt(sensorData.Accelerometer, sensorData.Gyroscope, sensorData.Magnetometer);
plotPerformance(oEst, groundTruth.Orientation, "Tuned Filter Error - Custom Cost Function");
```



Supporting Functions

plotPerformance Plot the orientation error

```
function plotPerformance(est, act, txt)
% Plot the orientation error in degrees in a new figure window.
figure;
plot(rad2deg(dist(est, act(1:10:end))));
ylabel("Degrees")
title(txt);
end
```

Magnetometer Calibration

Magnetometers detect magnetic field strength along a sensor's X,Y and Z axes. Accurate magnetic field measurements are essential for sensor fusion and the determination of heading and orientation.

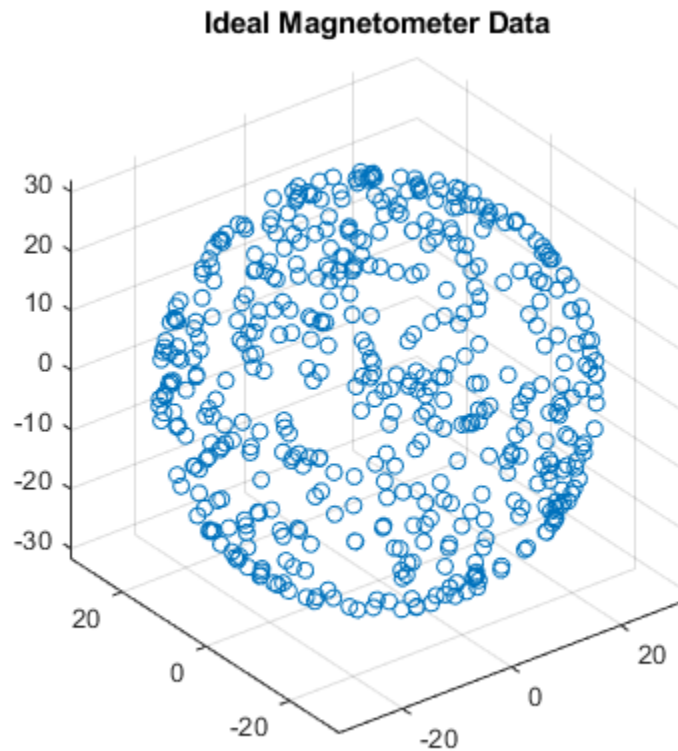
In order to be useful for heading and orientation computation, typical low cost MEMS magnetometers need to be calibrated to compensate for environmental noise and manufacturing defects.

Ideal Magnetometers

An ideal three-axis magnetometer measures magnetic field strength along orthogonal X, Y and Z axes. Absent any magnetic interference, magnetometer readings measure the Earth's magnetic field. If magnetometer measurements are taken as the sensor is rotated through all possible orientations, the measurements should lie on a sphere. The radius of the sphere is the magnetic field strength.

To generate magnetic field samples, use the `imuSensor` object. For these purposes it is safe to assume the angular velocity and acceleration are zero at each orientation.

```
N = 500;
rng(1);
acc = zeros(N,3);
av = zeros(N,3);
q = randrot(N,1); % uniformly distributed random rotations
imu = imuSensor('accel-mag');
[~,x] = imu(acc,av,q);
scatter3(x(:,1),x(:,2),x(:,3));
axis equal
title('Ideal Magnetometer Data');
```

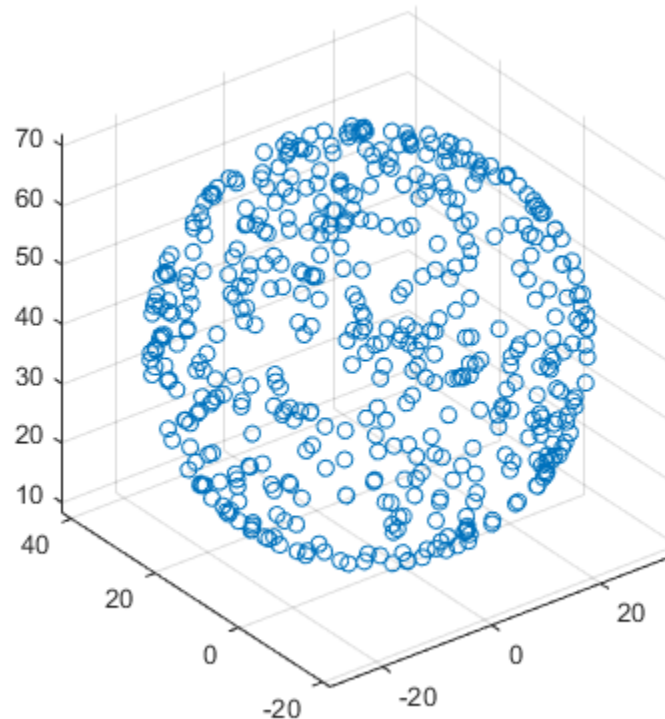


Hard Iron Effects

Noise sources and manufacturing defects degrade a magnetometer's measurement. The most striking of these are hard iron effects. Hard iron effects are stationary interfering magnetic noise sources. Often, these come from other metallic objects on the circuit board with the magnetometer. The hard iron effects shift the origin of the ideal sphere.

```
imu.Magnetometer.ConstantBias = [2 10 40];  
[~,x] = imu(acc,av,q);  
figure;  
scatter3(x(:,1),x(:,2),x(:,3));  
axis equal  
title('Magnetometer Data With a Hard Iron Offset');
```

Magnetometer Data With a Hard Iron Offset



Soft Iron Effects

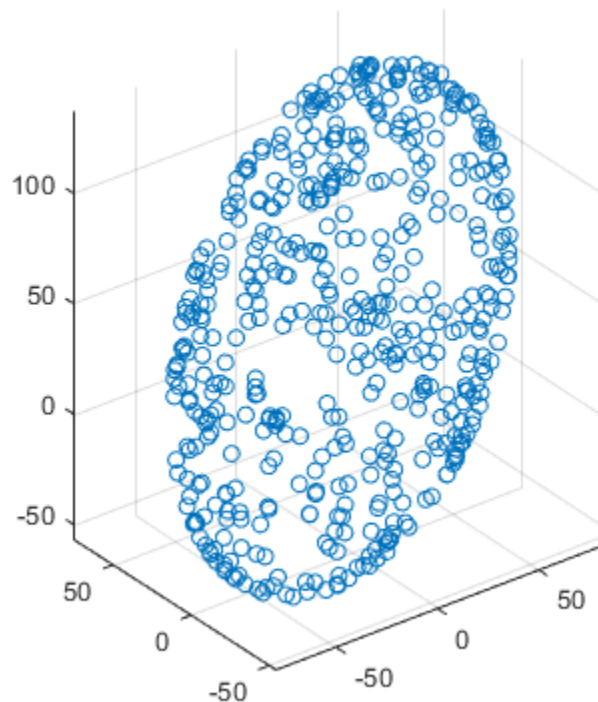
Soft iron effects are more subtle. They arise from objects near the sensor which distort the surrounding magnetic field. These have the effect of stretching and tilting the sphere of ideal measurements. The resulting measurements lie on an ellipsoid.

The soft iron magnetic field effects can be simulated by rotating the geomagnetic field vector of the IMU to the sensor frame, stretching it, and then rotating it back to the global frame.

```
nedmf = imu.MagneticField;
Rsoft = [2.5 0.3 0.5; 0.3 2 .2; 0.5 0.2 3];
soft = rotateframe(conj(q), rotateframe(q, nedmf) * Rsoft);

for ii=1:numel(q)
    imu.MagneticField = soft(ii,:);
    [~, x(ii,:)] = imu(acc(ii,:), av(ii,:), q(ii));
end
figure;
scatter3(x(:,1), x(:,2), x(:,3));
axis equal
title('Magnetometer Data With Hard and Soft Iron Effects');
```

Magnetometer Data With Hard and Soft Iron Effects



Correction Technique

The `magcal` function can be used to determine magnetometer calibration parameters that account for both hard and soft iron effects. Uncalibrated magnetometer data can be modeled as lying on an ellipsoid with equation

$$(x - b)R(x - b)^T = \beta^2$$

In this equation R is a 3-by-3 matrix, b is a 1-by-3 vector defining the ellipsoid center, x is a 1-by-3 vector of uncalibrated magnetometer measurements, and β is a scalar indicating the magnetic field strength. The above equation is the general form of a conic. For an ellipsoid, R must be positive definite. The `magcal` function uses a variety of solvers, based on different assumptions about R . In the `magcal` function, R can be assumed to be the identity matrix, a diagonal matrix, or a symmetric matrix.

The `magcal` function produces correction coefficients that take measurements which lie on an offset ellipsoid and transform them to lie on an ideal sphere, centered at the origin. The `magcal` function returns a 3-by-3 real matrix A and a 1-by-3 vector b . To correct the uncalibrated data compute

$$m = (x - b)A.$$

Here x is a 1-by-3 array of uncalibrated magnetometer measurements and m is the 1-by-3 array of corrected magnetometer measurements, which lie on a sphere. The matrix A has a determinant of 1 and is the matrix square root of R . Additionally, A has the same form as R : the identity, a diagonal, or

a symmetric matrix. Because these kinds of matrices cannot impart a rotation, the matrix A will not rotate the magnetometer data during correction.

The `magcal` function also returns a third output which is the magnetic field strength β . You can use the magnetic field strength to set the `ExpectedMagneticFieldStrength` property of `ahrsfilter`.

Using the `magcal` Function

Use the `magcal` function to determine calibration parameters that correct noisy magnetometer data. Create noisy magnetometer data by setting the `NoiseDensity` property of the `Magnetometer` property in the `imuSensor`. Use the rotated and stretched magnetic field in the variable `soft` to simulate soft iron effects.

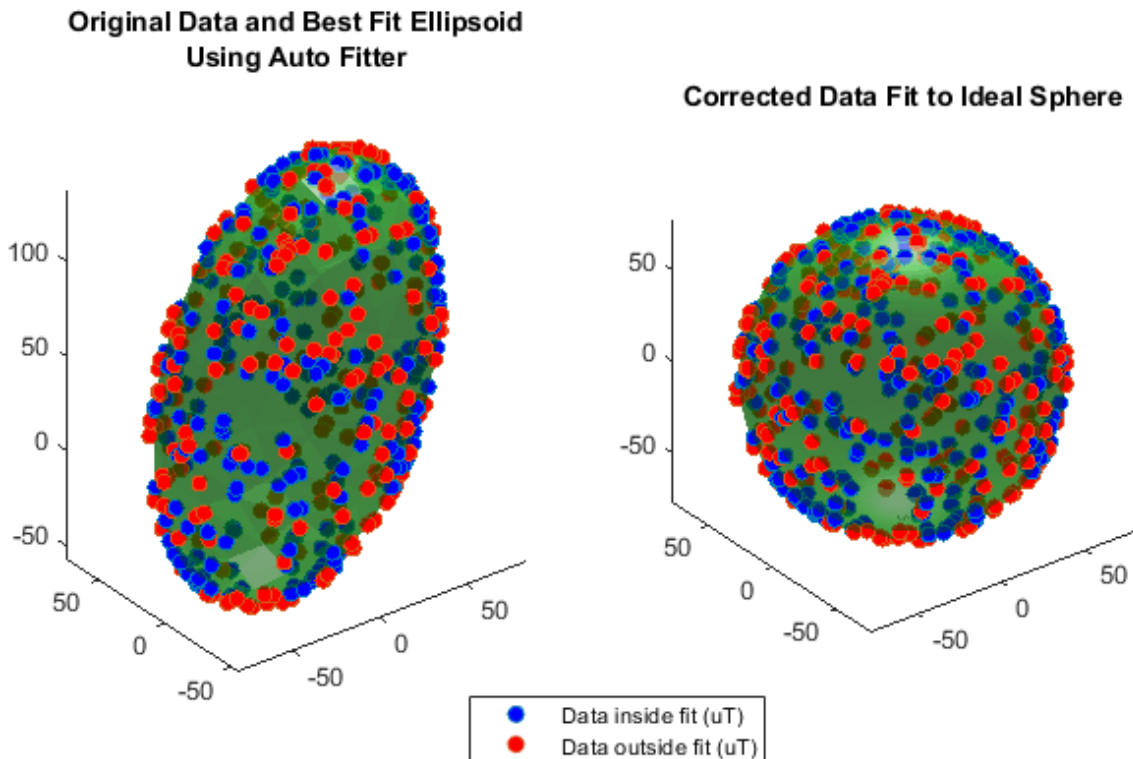
```
imu.Magnetometer.NoiseDensity = 0.08;
for ii=1:numel(q)
    imu.MagneticField = soft(ii,:);
    [~,x(ii,:)] = imu(acc(ii,:),av(ii,:),q(ii));
end
```

To find the A and b parameters which best correct the uncalibrated magnetometer data, simply call the function as:

```
[A,b,expMFS] = magcal(x);
xCorrected = (x-b)*A;
```

Plot the original and corrected data. Show the ellipsoid that best fits the original data. Show the sphere on which the corrected data should lie.

```
de = HelperDrawEllipsoid;
de.plotCalibrated(A,b,expMFS,x,xCorrected,'Auto');
```



The `magcal` function uses a variety of solvers to minimize the residual error. The residual error is the sum of the distances between the calibrated data and a sphere of radius `expMFS`.

$$E = \frac{1}{2\beta^2} \sqrt{\frac{\sum \|(x - b)A\|^2 - \beta^2}{N}}$$

```
r = sum(xCorrected.^2,2) - expMFS.^2;
E = sqrt(r.'*r./N)./(2*expMFS.^2);
fprintf('Residual error in corrected data : %.2f\n\n',E);
```

Residual error in corrected data : 0.01

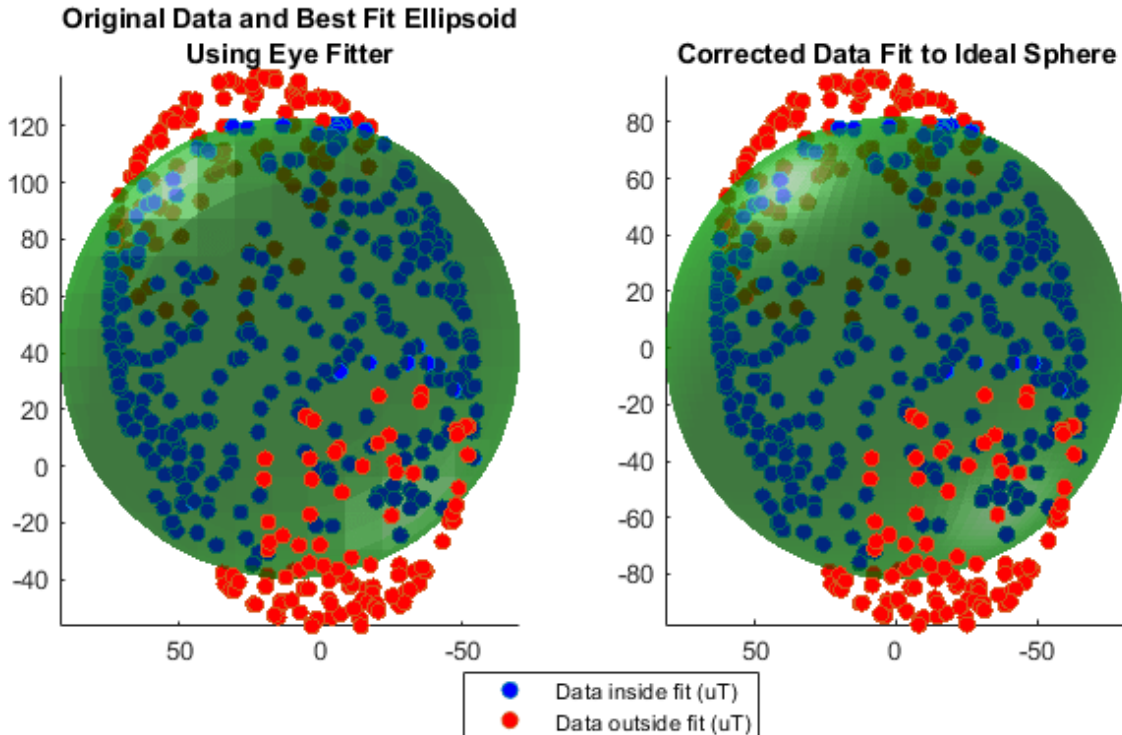
You can run the individual solvers if only some defects need to be corrected or to achieve a simpler correction computation.

Offset-Only Computation

Many MEMS magnetometers have registers within the sensor that can be used to compensate for the hard iron offset. In effect, the $(x-b)$ portion of the equation above happens on board the sensor. When only a hard iron offset compensation is needed, the A matrix effectively becomes the identity matrix. To determine the hard iron correction alone, the `magcal` function can be called this way:

```
[Aeye,beye,expMFSeye] = magcal(x,'eye');
xEyeCorrected = (x-beye)*Aeye;
[ax1,ax2] = de.plotCalibrated(Aeye,beye,expMFSeye,x,xEyeCorrected,'Eye');
```

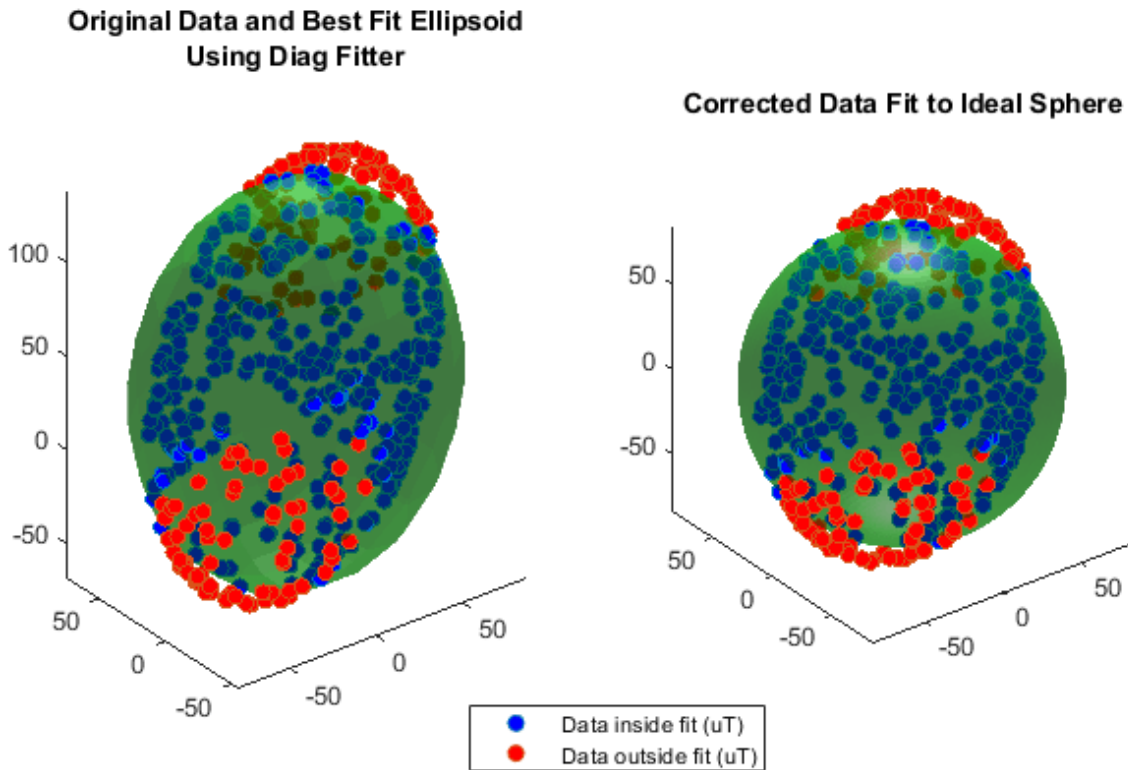
```
view(ax1, [-1 0 0]);
view(ax2, [-1 0 0]);
```



Hard Iron Compensation and Axis Scaling Computation

For many applications, treating the ellipsoid matrix as a diagonal matrix is sufficient. Geometrically, this means the ellipsoid of uncalibrated magnetometer data is approximated to have its semi-axes aligned with the coordinate system axes and a center offset from the origin. Though this is unlikely to be the actual characteristics of the ellipsoid, it reduces the correction equation to a single multiply and single subtract per axis.

```
[Adiag,bdiag,expMFSdiag] = magcal(x, 'diag');
xDiagCorrected = (x-bdiag)*Adiag;
[ax1,ax2] = de.plotCalibrated(Adiag,bdiag,expMFSdiag,x,xDiagCorrected,...
    'Diag');
```



Full Hard and Soft Iron Compensation

To force the `magcal` function to solve for an arbitrary ellipsoid and produce a dense, symmetric A matrix, call the function as:

```
[A,b] = magcal(x,'sym');
```

Auto Fit

The 'eye', 'diag', and 'sym' flags should be used carefully and the output values inspected. In some cases, there may be insufficient data for a high order ('diag' or 'sym') fit and a better set of correction parameters can be found using a simpler A matrix. The 'auto' fit option, which is the default, handles this situation.

Consider the case when insufficient data is used with a high order fitter.

```
xidx = x(:,3) > 100;
xpoor = x(xidx,:);
[Apoor,bpoor,mfspoor] = magcal(xpoor,'diag');
```

There is not enough data spread over the surface of the ellipsoid to achieve a good fit and proper calibration parameters with the 'diag' option. As a result, the `Apoor` matrix is complex.

```
disp(Apoor)
0.0000 + 0.4722i    0.0000 + 0.0000i    0.0000 + 0.0000i
0.0000 + 0.0000i    0.0000 + 0.5981i    0.0000 + 0.0000i
0.0000 + 0.0000i    0.0000 + 0.0000i    3.5407 + 0.0000i
```

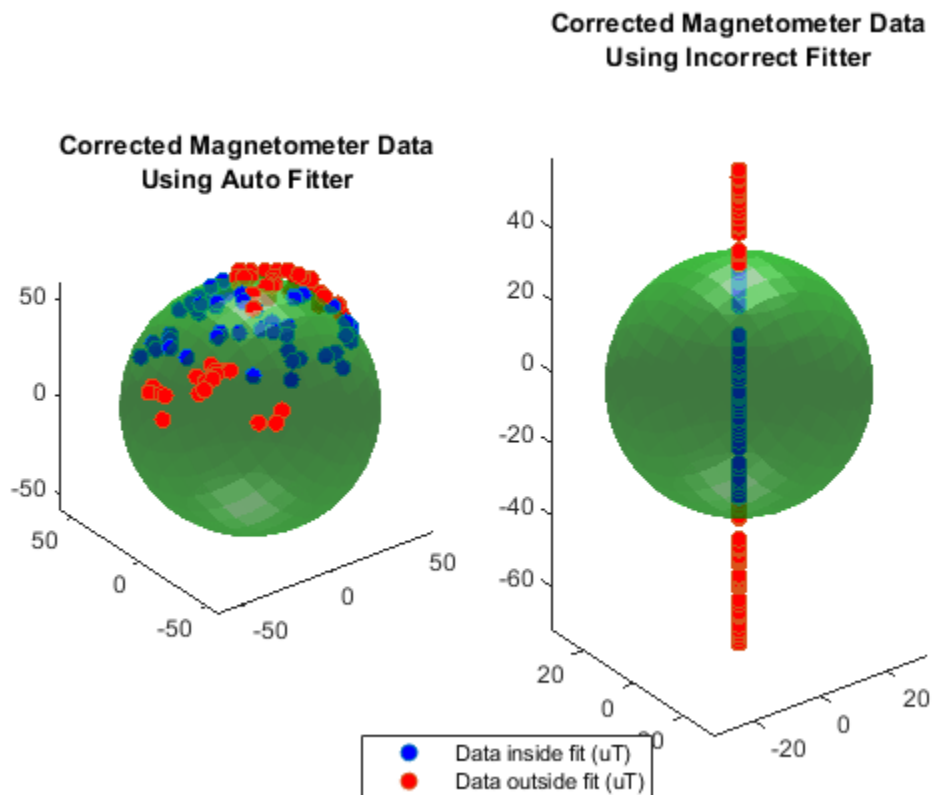
Using the 'auto' fit option avoids this problem and finds a simpler A matrix which is real, symmetric, and positive definite. Calling `magcal` with the 'auto' option string is the same as calling without any option string.

```
[Abest,bbest,mfsbest] = magcal(xpoor,'auto');
disp(Abest)
```

```
1    0    0
0    1    0
0    0    1
```

Comparing the results of using the 'auto' fitter and an incorrect, high order fitter show the perils of not examining the returned A matrix before correcting the data.

```
de.compareBest(Abest,bbest,mfsbest,Apoor,bpoor,mfspoer,xpoor);
```



Calling the `magcal` function with the 'auto' flag, which is the default, will try all possibilities of 'eye', 'diag' and 'sym' searching for the A and b which minimizes the residual error, keeps A real, and ensures R is positive definite and symmetric.

Conclusion

The `magcal` function can give calibration parameters to correct hard and soft iron offsets in a magnetometer. Calling the function with no option string, or equivalently the 'auto' option string, produces the best fit and covers most cases.

Wheel Encoder Error Sources

Explore the various error sources of wheel encoders and how they affect the wheel odometry estimate. After defining a ground truth trajectory, change parameters for wheel radius bias, wheel position noise, wheel slippage, and track width for the various wheel encoder objects. Notice the affects of changing these parameters on the output trajectory from the wheel encoder sensor models.

Ground Truth Trajectory

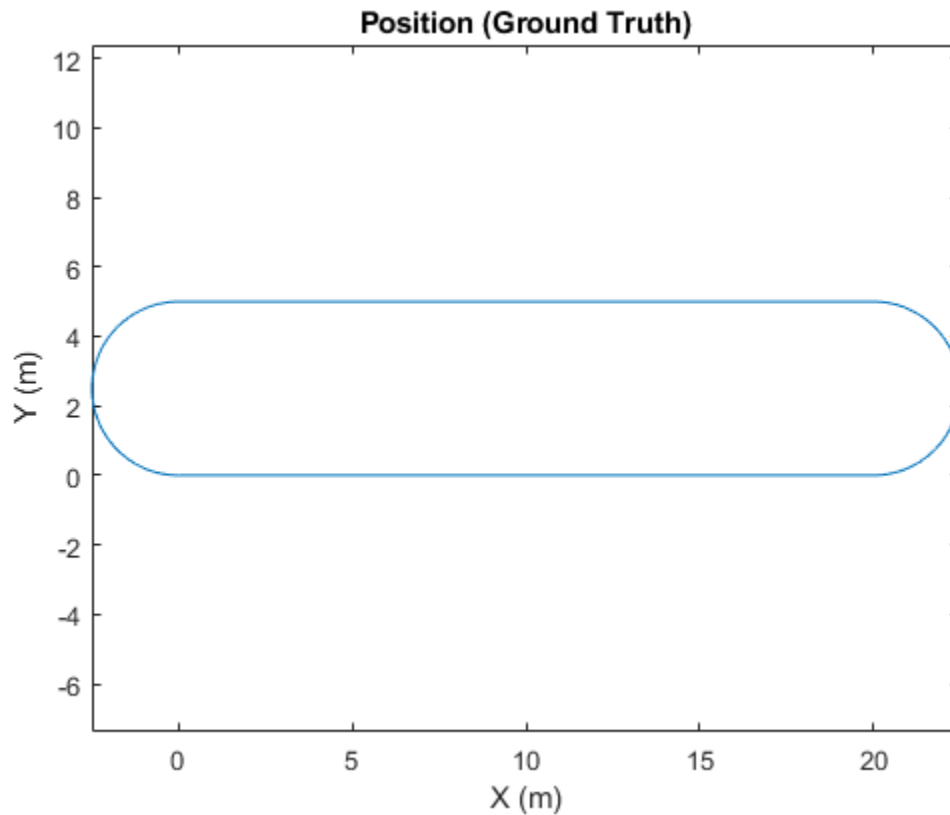
Create a ground truth trajectory to be used when examining the error parameters. Plot the trajectory.

```
Fs = 100;
wps = [0 0 0;
       20 0 0;
       20 5 0;
       0 5 0;
       0 0 0];
toa = cumsum([0 10 1.25*pi 10 1.25*pi]).';
vels = [2 0 0;
        2 0 0;
        -2 0 0;
        -2 0 0;
        2 0 0];

traj = waypointTrajectory(wps,...
    'SampleRate',Fs,'ReferenceFrame','ENU', ...
    'TimeOfArrival',toa,'Velocities',vels);

% Fetch pose values.
[pos,orient,vel,acc,angvel] = lookupPose(traj,toa(1):1/Fs:toa(end));
angvelBody = rotateframe(orient,angvel);

% Plot ground truth position.
figure
plot(pos(:,1),pos(:,2))
title('Position (Ground Truth)')
xlabel('X (m)')
ylabel('Y (m)')
axis equal
```



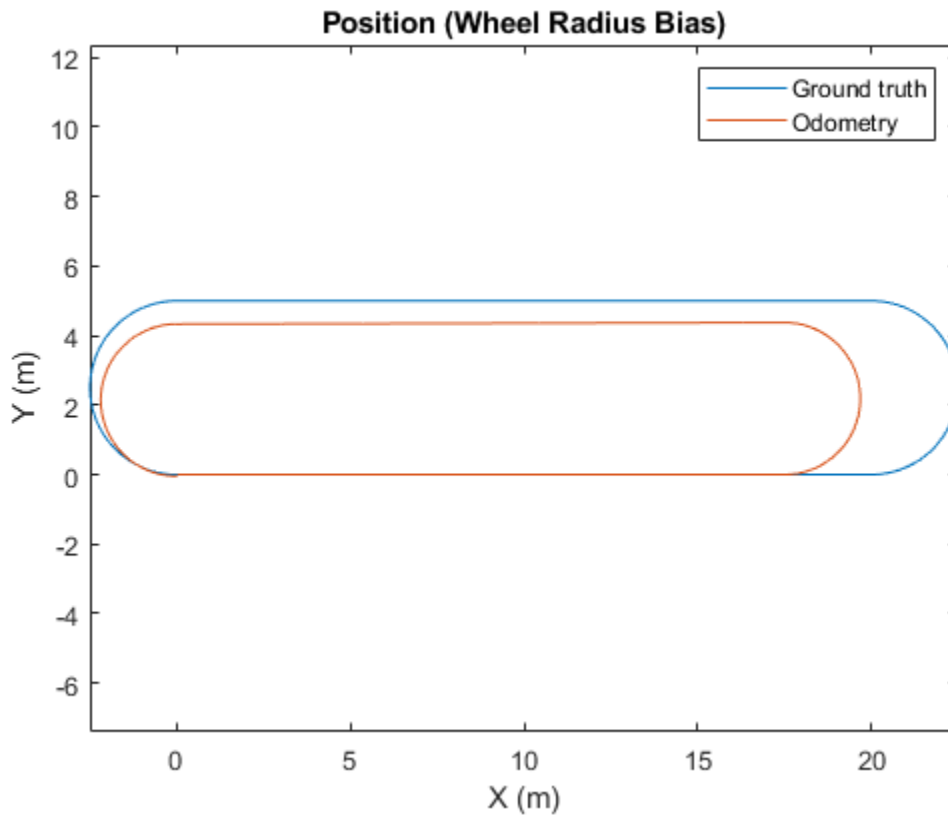
Bias in the Wheel Radius

Create a wheel encoder sensor for a unicycle model as a `wheelEncoderUnicycle` object. Specify a non-zero `WheelRadiusBias` and examine how it affects the odometry estimate. Specifying a positive bias causes the odometry algorithm to underestimate the circumference of the wheel. This results in the odometry estimating a smaller distance traveled.

```
encoder = wheelEncoderUnicycle;
encoder.WheelRadiusBias = 0.05;
odom = wheelEncoderOdometryUnicycle(encoder);

ticks = encoder(vel, angvel, orient);
estPose = odom(ticks, angvelBody(:,3));

% Plot ground truth and estimated positions.
figure
plot(pos(:,1),pos(:,2),estPose(:,1),estPose(:,2))
title('Position (Wheel Radius Bias)')
xlabel('X (m)')
ylabel('Y (m)')
legend('Ground truth','Odometry')
axis equal
```



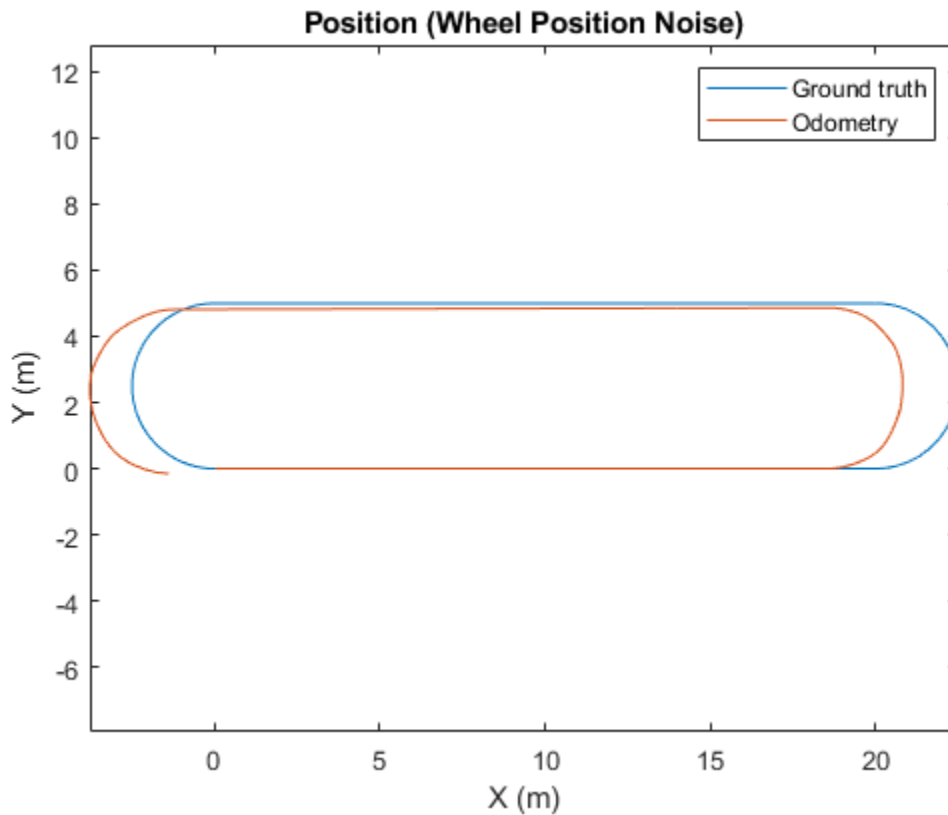
Noise in the Wheel Position Measurement

Specify a non-zero `WheelPositionAccuracy` and examine how it affects the odometry estimate. This noise adds random deviations to the measured ticks from the wheel encoder.

```
encoder = wheelEncoderUnicycle;
encoder.WheelPositionAccuracy = 0.1;
% Use a local random stream to reproduce results.
encoder.RandomStream = 'mt19937ar with seed';
odom = wheelEncoderOdometryUnicycle(encoder);

ticks = encoder(vel,angvel,orient);
estPose = odom(ticks,angvelBody(:,3));

% Plot ground truth and estimated positions.
figure
plot(pos(:,1),pos(:,2),estPose(:,1),estPose(:,2))
title('Position (Wheel Position Noise)')
xlabel('X (m)')
ylabel('Y (m)')
legend('Ground truth', 'Odometry')
axis equal
```

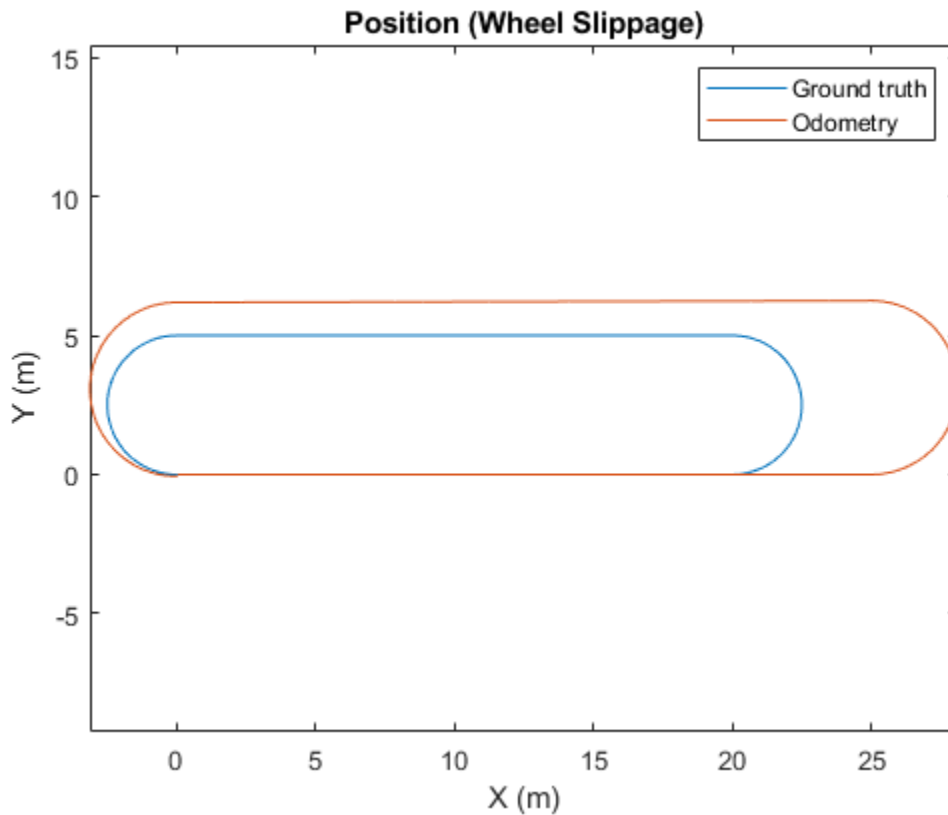
Wheel Slippage and Skidding

Specify a non-zero `SlipRatio` and examine how it affects the odometry estimate. Specifying a value greater than zero simulates wheel slippage. This slippage results in the odometry estimating a larger distance traveled. A negative value for slip ratio indicates skidding.

```
encoder = wheelEncoderUnicycle;
encoder.SlipRatio = 0.25;
odom = wheelEncoderOdometryUnicycle(encoder);

ticks = encoder(vel, angvel, orient);
estPose = odom(ticks, angvelBody(:,3));

% Plot ground truth and estimated positions.
figure
plot(pos(:,1),pos(:,2),estPose(:,1),estPose(:,2))
title('Position (Wheel Slippage)')
xlabel('X (m)')
ylabel('Y (m)')
legend('Ground truth', 'Odometry')
axis equal
```



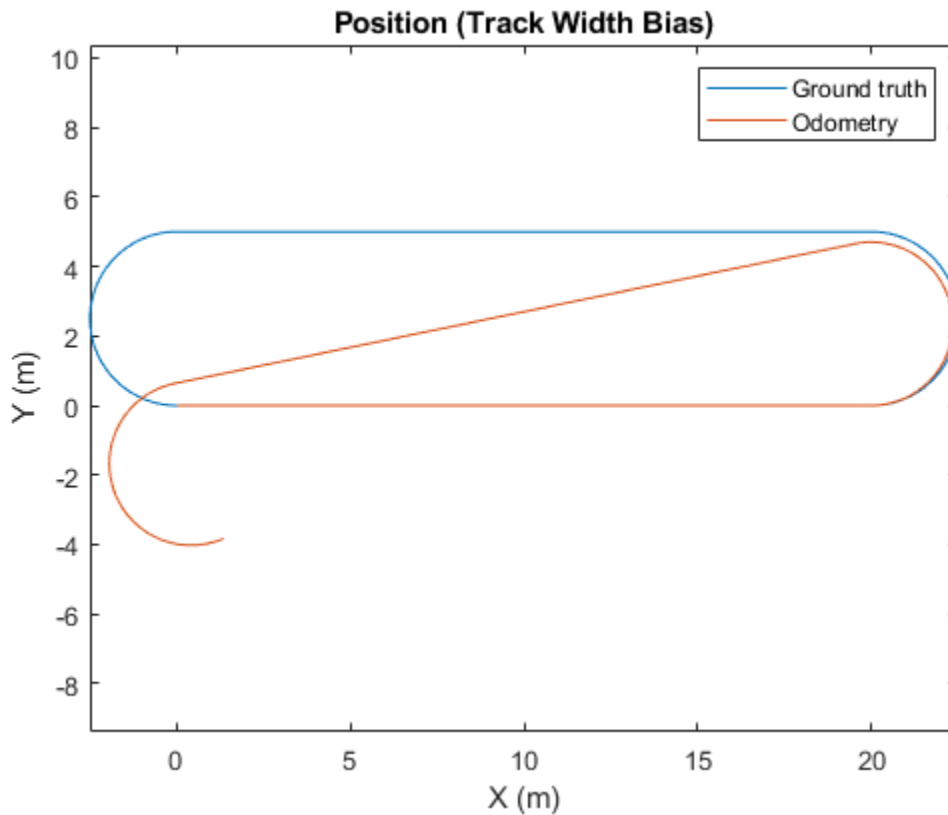
Bias in the Track Width

Specify a non-zero `TrackWidthBias` and examine how it affects the odometry estimate. Specifying a positive bias will cause the odometry algorithm to overestimate the turning angle of the vehicle. This overestimation of turning results in drift accumulating in the odometry estimate at turns. For this scenario, a vehicle with an axle is needed, so use the `wheelEncoderDifferentialDrive` object.

```
encoder = wheelEncoderDifferentialDrive;
encoder.TrackWidthBias = 0.1;
odom = wheelEncoderOdometryDifferentialDrive(encoder);

ticks = encoder(vel,angvel,orient);
estPose = odom(ticks);

% Plot ground truth and estimated positions.
figure
plot(pos(:,1),pos(:,2), estPose(:,1), estPose(:,2))
title('Position (Track Width Bias)')
xlabel('X (m)')
ylabel('Y (m)')
legend('Ground truth', 'Odometry')
axis equal
```



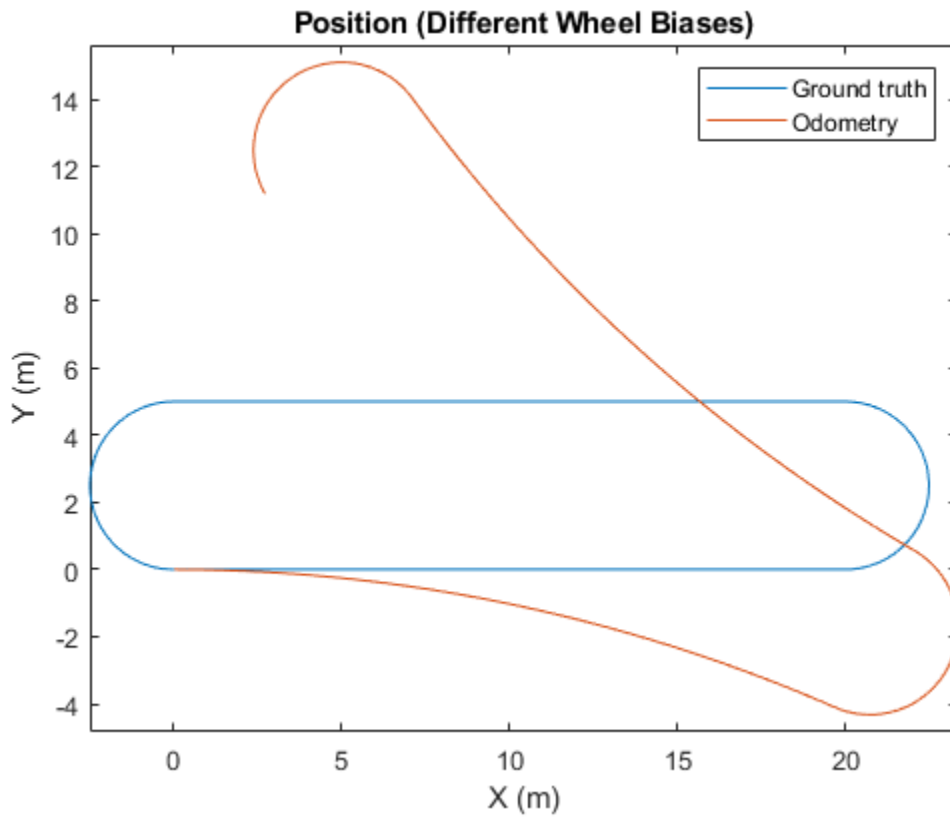
Differing Biases in the Wheels

Specify different non-zero values for the `WheelRadiusBias` and examine how it affects the odometry estimate. Specifying different biases causes the odometry estimate to drift throughout the entire trajectory. For this scenario, two wheels are needed, so use the `wheelEncoderDifferentialDrive` object.

```
encoder = wheelEncoderDifferentialDrive;
encoder.WheelRadiusBias = [-0.01, 0.001];
odom = wheelEncoderOdometryDifferentialDrive(encoder);

ticks = encoder(vel, angvel, orient);
estPose = odom(ticks);

% Plot ground truth and estimated positions.
figure
plot(pos(:,1),pos(:,2), estPose(:,1), estPose(:,2))
title('Position (Different Wheel Biases)')
xlabel('X (m)')
ylabel('Y (m)')
legend('Ground truth', 'Odometry')
axis equal
```



Remove Bias from Angular Velocity Measurement

This example shows how to remove gyroscope bias from an IMU using `imufilter`.

Use `kinematicTrajectory` to create a trajectory with two parts. The first part has a constant angular velocity about the *y*- and *z*-axes. The second part has a varying angular velocity in all three axes.

```
duration = 60*8;
fs = 20;
numSamples = duration * fs;
rng('default') % Seed the RNG to reproduce noisy sensor measurements.

initialAngVel = [0,0.5,0.25];
finalAngVel = [-0.2,0.6,0.5];
constantAngVel = repmat(initialAngVel,floor(numSamples/2),1);
varyingAngVel = [linspace(initialAngVel(1), finalAngVel(1), ceil(numSamples/2)).', ...
    linspace(initialAngVel(2), finalAngVel(2), ceil(numSamples/2)).', ...
    linspace(initialAngVel(3), finalAngVel(3), ceil(numSamples/2)).'];

angVelBody = [constantAngVel; varyingAngVel];
accBody = zeros(numSamples,3);

traj = kinematicTrajectory('SampleRate',fs);

[~,qNED,~,accNED,angVelNED] = traj(accBody,angVelBody);
```

Create an `imuSensor System` object™, `IMU`, with a nonideal gyroscope. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation.

```
IMU = imuSensor('accel-gyro', ...
    'Gyroscope',gyroparams('RandomWalk',0.003,'ConstantBias',0.3), ...
    'SampleRate',fs);

[accelReadings, gyroReadingsBody] = IMU(accNED,angVelNED,qNED);
```

Create an `imufilter System` object, `fuse`. Call `fuse` with the modeled accelerometer readings and gyroscope readings.

```
fuse = imufilter('SampleRate',fs, 'GyroscopeDriftNoise', 1e-6);

[~,angVelBodyRecovered] = fuse(accelReadings,gyroReadingsBody);
```

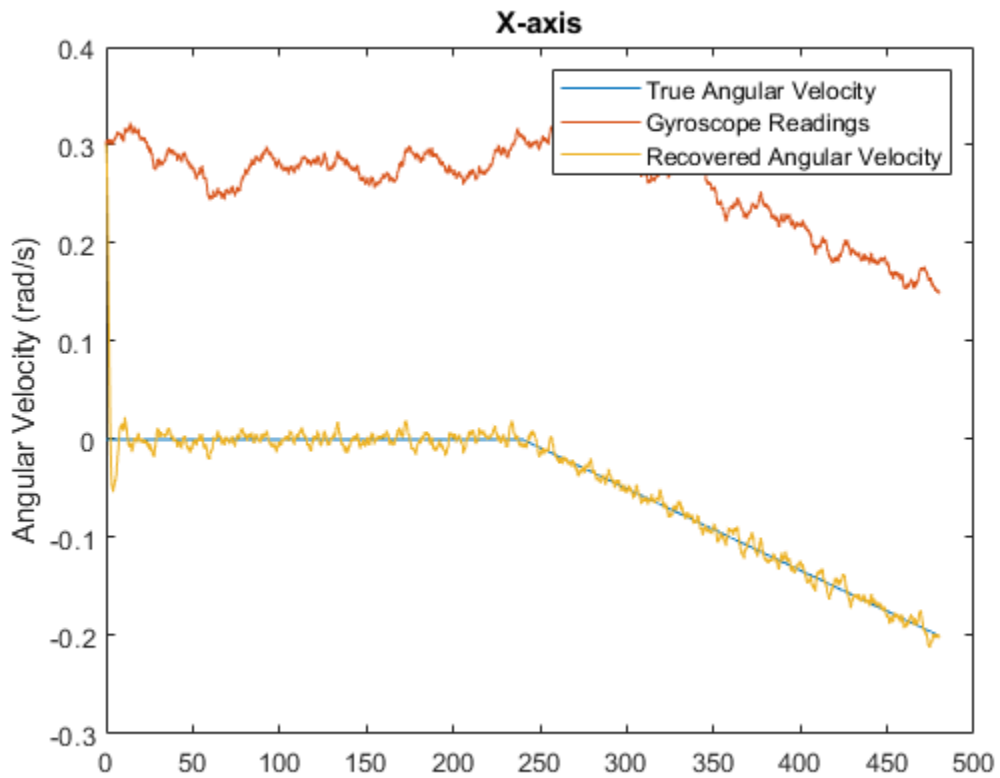
Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

The angular velocity returned from the `imufilter` compensates for the effect of the gyroscope bias over time and converges to the true angular velocity.

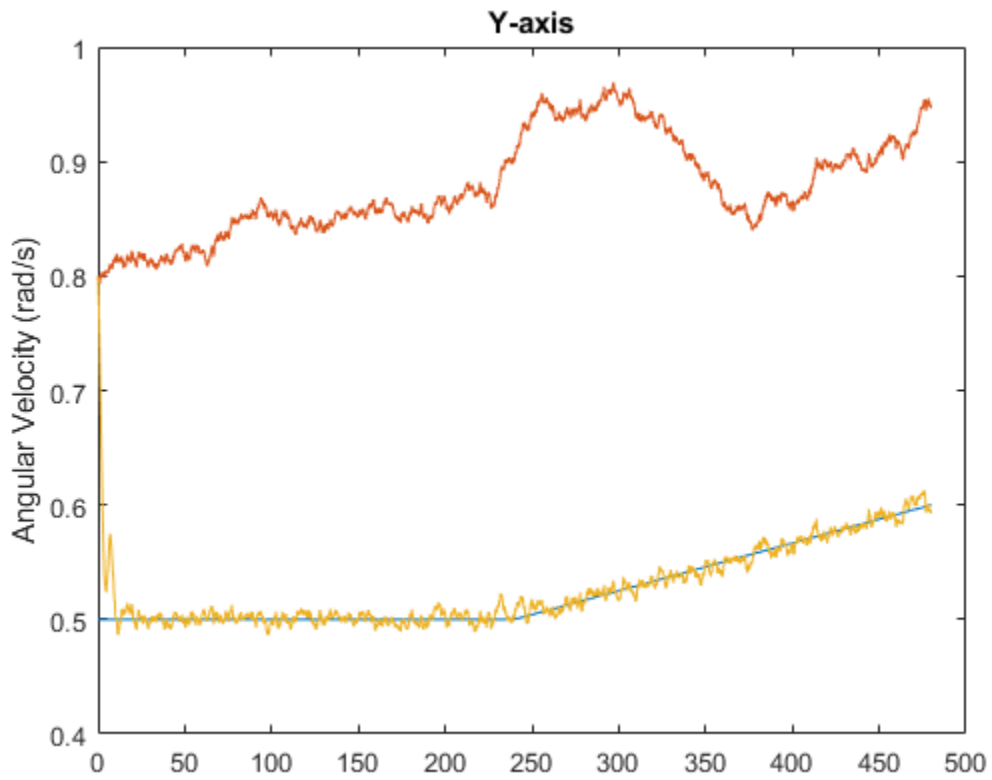
```
time = (0:numSamples-1)/fs;

figure(1)
plot(time,angVelBody(:,1), ...
    time,gyroReadingsBody(:,1), ...
    time,angVelBodyRecovered(:,1))
title('X-axis')
legend('True Angular Velocity', ...
```

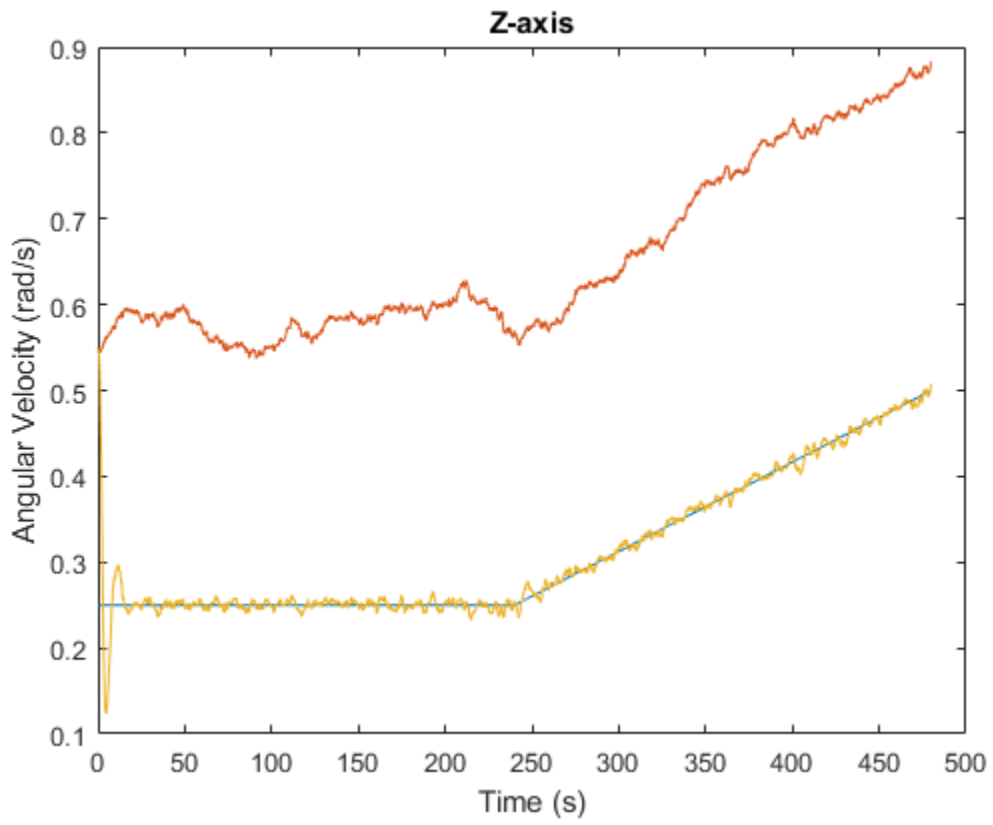
```
'Gyroscope Readings', ...
'Recovered Angular Velocity')
ylabel('Angular Velocity (rad/s)')
```



```
figure(2)
plot(time,angVelBody(:,2), ...
      time,gyroReadingsBody(:,2), ...
      time,angVelBodyRecovered(:,2))
title('Y-axis')
ylabel('Angular Velocity (rad/s)')
```



```
figure(3)
plot(time,angVelBody(:,3), ...
      time,gyroReadingsBody(:,3), ...
      time,angVelBodyRecovered(:,3))
title('Z-axis')
ylabel('Angular Velocity (rad/s)')
xlabel('Time (s)')
```



Detect Multipath GPS Reading Errors Using Residual Filtering in Inertial Sensor Fusion

This example shows how to use the `residualgps` object function and residual filtering to detect when new sensor measurements may not be consistent with the current filter state.

Load Trajectory and Sensor Data

Load the MAT-file `loggedDataWithMultipath.mat`. This file contains simulated IMU and GPS data as well as the ground truth position and orientation of a circular trajectory. The GPS data contains errors due to multipath errors in one section of the trajectory. These errors were modelled by adding white noise to the GPS data to simulate the effects of an urban canyon.

```
load('loggedDataWithMultipath.mat', ...
     'imuFs', 'accel', 'gyro', ... % IMU readings
     'gpsFs', 'lla', 'gpsVel', ... % GPS readings
     'truePosition', 'trueOrientation', ... % Ground truth pose
     'localOrigin', 'initialState', 'multipathAngles')

% Number of IMU samples per GPS sample.
imuSamplesPerGPS = (imuFs/gpsFs);

% First and last indices corresponding to multipath errors.
multipathIndices = [1850 2020];
```

Fusion Filter

Create two pose estimation filters using the `insfilterNonholonomic` object. Use one filter to process all the sensor readings. Use the other filter to only process the sensor readings that are not considered outliers.

```
% Create filters.

% Use this filter to only process sensor readings that are not detected as
% outliers.
gndFusionWithDetection = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
     'IMUSampleRate', imuFs, ...
     'ReferenceLocation', localOrigin, ...
     'DecimationFactor', 2);

% Use this filter to process all sensor readings, regardless of whether or
% not they are outliers.
gndFusionNoDetection = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
     'IMUSampleRate', imuFs, ...
     'ReferenceLocation', localOrigin, ...
     'DecimationFactor', 2);

% GPS measurement noises.
Rvel = 0.01;
Rpos = 1;

% The dynamic model of the ground vehicle for this filter assumes there is
% no side slip or skid during movement. This means that the velocity is
% constrained to only the forward body axis. The other two velocity axis
% readings are corrected with a zero measurement weighted by the
% |ZeroVelocityConstraintNoise| parameter.
gndFusionWithDetection.ZeroVelocityConstraintNoise = 1e-2;
```

```
gndFusionNoDetection.ZeroVelocityConstraintNoise = 1e-2;

% Process noises.
gndFusionWithDetection.GyroscopeNoise = 4e-6;
gndFusionWithDetection.GyroscopeBiasNoise = 4e-14;
gndFusionWithDetection.AccelerometerNoise = 4.8e-2;
gndFusionWithDetection.AccelerometerBiasNoise = 4e-14;
gndFusionNoDetection.GyroscopeNoise = 4e-6;
gndFusionNoDetection.GyroscopeBiasNoise = 4e-14;
gndFusionNoDetection.AccelerometerNoise = 4.8e-2;
gndFusionNoDetection.AccelerometerBiasNoise = 4e-14;

% Initial filter states.
gndFusionWithDetection.State = initialState;
gndFusionNoDetection.State = initialState;

% Initial error covariance.
gndFusionWithDetection.StateCovariance = 1e-9*ones(16);
gndFusionNoDetection.StateCovariance = 1e-9*ones(16);
```

Initialize Scopes

The `HelperPoseViewer` scope allows a 3-D visualization comparing the filter estimate and ground truth. Using multiple scopes can slow the simulation. To disable the scopes, set the corresponding logical variable to `false`.

```
usePoseView = true; % Turn on the 3D pose viewer

if usePoseView
    [viewerWithDetection, viewerNoDetection, annoHandle] ...
        = helperCreatePoseViewers(initialState, multipathAngles);
end
```

Simulation Loop

The main simulation loop is a `for` loop with a nested `for` loop. The first loop executes at the `gpsFs`, which is the GPS measurement rate. The nested loop executes at the `imuFs`, which is the IMU sample rate. Each scope is updated at the IMU sample rate.

```
numSamples = numel(trueOrientation);
numGPSSamples = numSamples/imuSamplesPerGPS;

% Log data for final metric computation.
estPositionNoCheck = zeros(numSamples, 3);
estOrientationNoCheck = quaternion.zeros(numSamples, 1);

estPosition = zeros(numSamples, 3);
estOrientation = quaternion.zeros(numSamples, 1);

% Threshold for outlier residuals.
residualThreshold = 6;

idx = 0;
for sampleIdx = 1:numGPSSamples
    % Predict loop at IMU update frequency.
    for i = 1:imuSamplesPerGPS
        idx = idx + 1;
```

```

% Use the predict method to estimate the filter state based
% on the accelData and gyroData arrays.
predict(gndFusionWithDetection, accel(idx,:), gyro(idx,:));

predict(gndFusionNoDetection, accel(idx,:), gyro(idx,:));

% Log the estimated orientation and position.
[estPositionNoCheck(idx,:), estOrientationNoCheck(idx,:)] ...
    = pose(gndFusionWithDetection);

[estPosition(idx,:), estOrientation(idx,:)] ...
    = pose(gndFusionNoDetection);

% Update the pose viewer.
if usePoseView
    viewerWithDetection(estPositionNoCheck(idx,:), ...
        estOrientationNoCheck(idx,:), ...
        truePosition(idx,:), trueOrientation(idx,:));

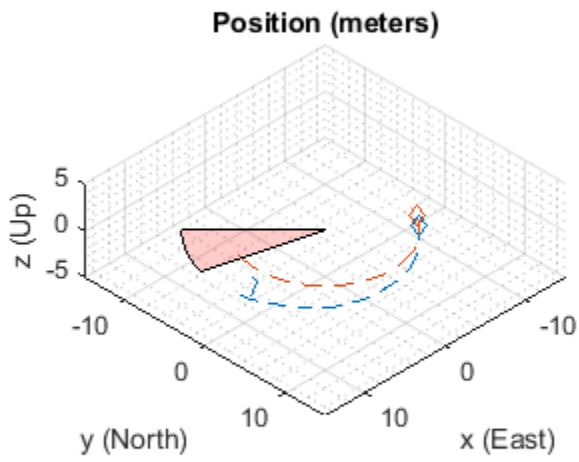
    viewerNoDetection(estPosition(idx,:), ...
        estOrientation(idx,:), truePosition(idx,:), ...
        trueOrientation(idx,:));
end
end

% This next section of code runs at the GPS sample rate.

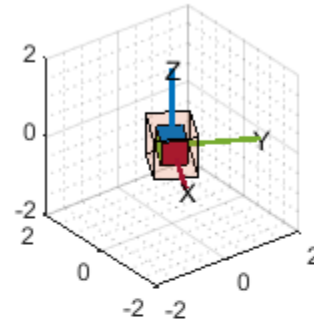
% Update the filter states based on the GPS data.
fusegps(gndFusionWithDetection, lla(sampleIdx,:), Rpos, ...
    gpsVel(sampleIdx,:), Rvel);

% Check the normalized residual of the current GPS reading. If the
% value is too large, it is considered an outlier and disregarded.
[res, resCov] = residualgps(gndFusionNoDetection, lla(sampleIdx,:), ...
    Rpos, gpsVel(sampleIdx,:), Rvel);
normalizedRes = res(1:3) ./ sqrt( diag(resCov(1:3,1:3)).' );
if (all(abs(normalizedRes) <= residualThreshold))
    % Update the filter states based on the GPS data.
    fusegps(gndFusionNoDetection, lla(sampleIdx,:), Rpos, ...
        gpsVel(sampleIdx,:), Rvel);
    if usePoseView
        set(annoHandle, 'String', 'Outlier status: none', ...
            'EdgeColor', 'k');
    end
else
    if usePoseView
        set(annoHandle, 'String', 'Outlier status: detected', ...
            'EdgeColor', 'r');
    end
end
end
end

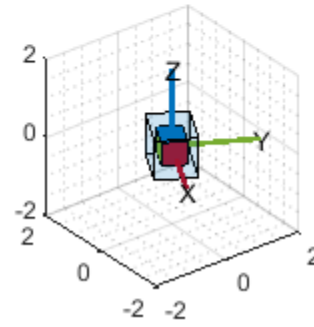
```

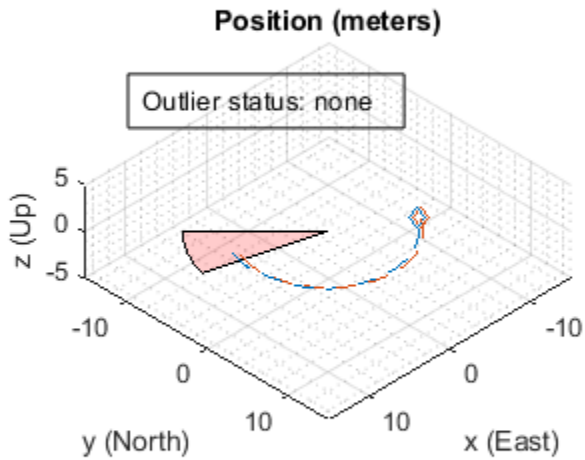


Orientation - Ground Truth

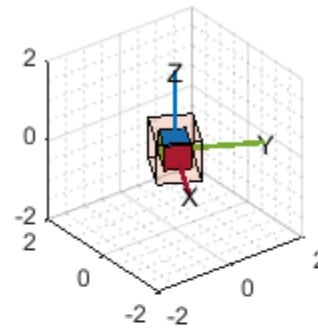


Orientation - Estimated

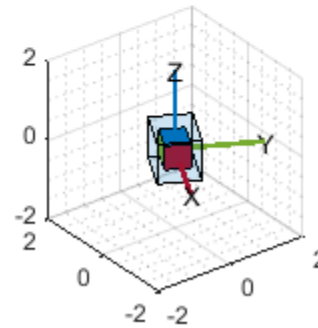




Orientation - Ground Truth



Orientation - Estimated



Error Metric Computation

Calculate the position error for both filter estimates. There is an increase in the position error in the filter that does not check for any outliers in the GPS measurements.

```
% Calculate position errors.
posdNoCheck = estPositionNoCheck - truePosition;
posd = estPosition - truePosition;

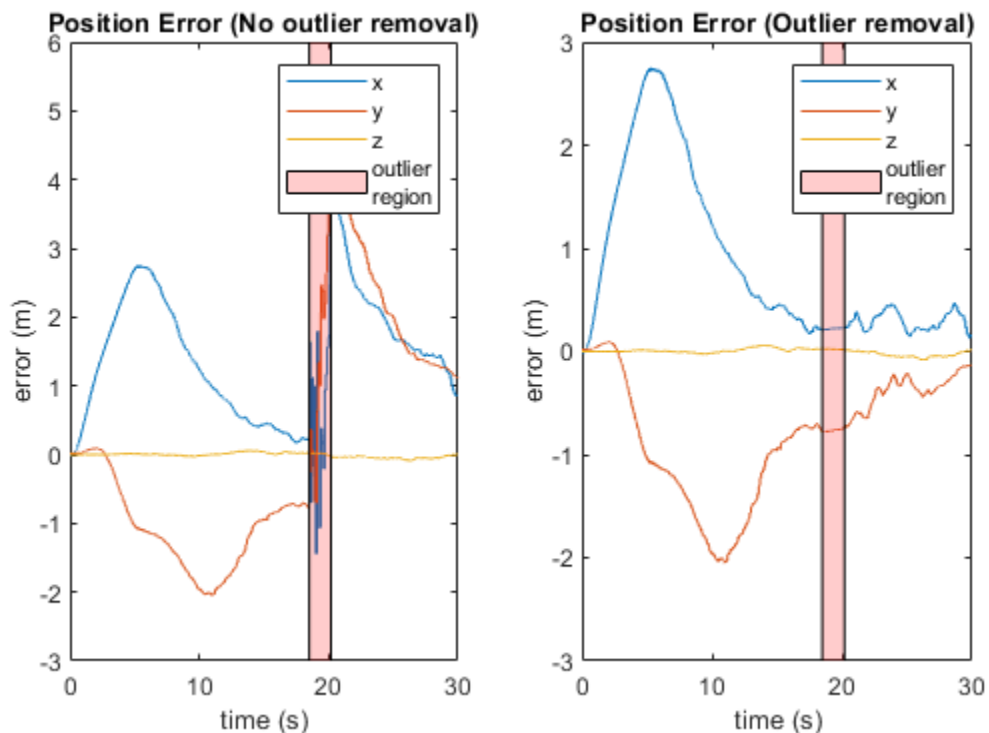
% Plot results.
t = (0:size(posd,1)-1).' ./ imuFs;
figure('Units', 'normalized', 'Position', [0.2615 0.2833 0.4552 0.3700])
subplot(1, 2, 1)
plot(t, posdNoCheck)
ax = gca;
yLims = get(ax, 'YLim');
hold on
mi = multipathIndices;
fill([t(mi(1)), t(mi(1)), t(mi(2)), t(mi(2))], [7 -5 -5 7], ...
    [1 0 0], 'FaceAlpha', 0.2);
set(ax, 'YLim', yLims);
title('Position Error (No outlier removal)')
xlabel('time (s)')
ylabel('error (m)')
legend('x', 'y', 'z', sprintf('outlier\nregion'))

subplot(1, 2, 2)
```

```

plot(t, posd)
ax = gca;
yLims = get(ax, 'YLim');
hold on
mi = multipathIndices;
fill([t(mi(1)), t(mi(1)), t(mi(2)), t(mi(2))], [7 -5 -5 7], ...
     [1 0 0], 'FaceAlpha', 0.2);
set(ax, 'YLim', yLims);
title('Position Error (Outlier removal)')
xlabel('time (s)')
ylabel('error (m)')
legend('x', 'y', 'z', sprintf('outlier\nregion'))

```



Conclusion

The `residualgps` object function can be used to detect potential outliers in sensor measurements before using them to update the filter states of the `insfilterNonholonomic` object. The other pose estimation filter objects such as, `insfilterMARG`, `insfilterAsync`, and `insfilterErrorState` also have similar object functions to calculate sensor measurement residuals.

Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter

This example shows how to fuse data from a 3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer (together commonly referred to as a MARG sensor for Magnetic, Angular Rate, and Gravity), and 1-axis altimeter to estimate orientation and height.

Simulation Setup

This simulation processes sensor data at multiple rates. The IMU (accelerometer and gyroscope) typically runs at the highest rate. The magnetometer generally runs at a lower rate than the IMU, and the altimeter runs at the lowest rate. Changing the sample rates causes parts of the fusion algorithm to run more frequently and can affect performance.

```
% Set the sampling rate for IMU sensors, magnetometer, and altimeter.
imuFs = 100;
altFs = 10;
magFs = 25;
imuSamplesPerAlt = fix(imuFs/altFs);
imuSamplesPerMag = fix(imuFs/magFs);

% Set the number of samples to simulate.
N = 1000;

% Construct object for other helper functions.
hfunc = Helper10AxisFusion;
```

Define Trajectory

The sensor body rotates about all three axes while oscillating in position vertically. The oscillations increase in magnitude as the simulation continues.

```
% Define the initial state of the sensor body
initPos = [0, 0, 0];      % initial position (m)
initVel = [0, 0, -1];    % initial linear velocity (m/s)
initOrient = ones(1, 'quaternion');
```

```
% Define the constant angular velocity for rotating the sensor body
% (rad/s).
angVel = [0.34 0.2 0.045];
```

```
% Define the acceleration required for simple oscillating motion of the
% sensor body.
fc = 0.2;
t = 0:1/imuFs:(N-1)/imuFs;
a = 1;
oscMotionAcc = sin(2*pi*fc*t);
oscMotionAcc = hfunc.growAmplitude(oscMotionAcc);
```

```
% Construct the trajectory object
traj = kinematicTrajectory('SampleRate', imuFs, ...
    'Velocity', initVel, ...
    'Position', initPos, ...
    'Orientation', initOrient);
```

Sensor Configuration

The accelerometer, gyroscope and magnetometer are simulated using `imuSensor`. The altimeter is modeled using the `altimeterSensor`. The values used in the sensor configurations correspond to real MEMS sensor values.

```
imu = imuSensor('accel-gyro-mag', 'SampleRate', imuFs);

% Accelerometer
imu.Accelerometer.MeasurementRange = 19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.ConstantBias = 0.19;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
imu.Gyroscope.ConstantBias = deg2rad(3.125);
imu.Gyroscope.AxesMisalignment = 1.5;
imu.Gyroscope.NoiseDensity = deg2rad(0.025);

% Magnetometer
imu.Magnetometer.MeasurementRange = 1000;
imu.Magnetometer.Resolution = 0.1;
imu.Magnetometer.ConstantBias = 100;
imu.Magnetometer.NoiseDensity = 0.3/sqrt(50);

% altimeter
altimeter = altimeterSensor('UpdateRate', altFs, 'NoiseDensity', 2*0.1549);
```

Fusion Filter

Construct an `ahrs10filter` and configure.

```
fusionfilt = ahrs10filter;
fusionfilt.IMUSampleRate = imuFs;
```

Set initial values for the fusion filter.

```
initstate = zeros(18,1);
initstate(1:4) = compact(initOrient);
initstate(5) = initPos(3);
initstate(6) = initVel(3);
initstate(7:9) = imu.Gyroscope.ConstantBias/imuFs;
initstate(10:12) = imu.Accelerometer.ConstantBias/imuFs;
initstate(13:15) = imu.MagneticField;
initstate(16:18) = imu.Magnetometer.ConstantBias;
fusionfilt.State = initstate;
```

Initialize the state covariance matrix of the fusion filter. The ground truth is used for initial states, so there should be little error in the estimates.

```
icv = diag([1e-8*[1 1 1 1 1 1 1], 1e-3*ones(1,11)]);
fusionfilt.StateCovariance = icv;
```

Magnetometer and altimeter measurement noises are the observation noises associated with the sensors used by the internal Kalman filter in the `ahrs10filter`. These values would normally come from a sensor datasheet.


```
magNoise = 2*(imu.Magnetometer.NoiseDensity(1).^2)*imuFs;
altimeterNoise = 2*(altimeter.NoiseDensity).^2 * altFs;
```

Filter process noises are used to tune the filter to desired performance.

```
fusionfilt.AccelerometerNoise = [1e-1 1e-1 1e-4];
fusionfilt.AccelerometerBiasNoise = 1e-8;
fusionfilt.GeomagneticVectorNoise = 1e-12;
fusionfilt.MagnetometerBiasNoise = 1e-12;
fusionfilt.GyroscopeNoise = 1e-12;
```

Additional Simulation Option : Viewer

By default, this simulation plots the estimation errors at the end of the simulation. To view both the estimated position and orientation along with the ground truth as the simulation runs, set the `usePoseViewer` variable to `true`.

```
usePoseViewer = false;
```

Simulation Loop

```
q = initOrient;
firstTime = true;

actQ = zeros(N,1, 'quaternion');
expQ = zeros(N,1, 'quaternion');
actP = zeros(N,1);
expP = zeros(N,1);

for ii = 1: N
    % Generate a new set of samples from the trajectory generator
    accBody = rotateframe(q, [0 0 +oscMotionAcc(ii)]);
    omgBody = rotateframe(q, angVel);
    [pos, q, vel, acc] = traj(accBody, omgBody);

    % Feed the current position and orientation to the imuSensor object
    [accel, gyro, mag] = imu(acc, omgBody, q);
    fusionfilt.predict(accel, gyro);

    % Fuse magnetometer samples at the magnetometer sample rate
    if ~mod(ii,imuSamplesPerMag)
        fusemag(fusionfilt, mag, magNoise);
    end

    % Sample and fuse the altimeter output at the altimeter sample rate
    if ~mod(ii,imuSamplesPerAlt)
        altHeight = altimeter(pos);

        % Use the |fusealtimeter| method to update the fusion filter with
        % the altimeter output.
        fusealtimeter(fusionfilt,altHeight,altimeterNoise);
    end

    % Log the actual orientation and position
    [actP(ii), actQ(ii)] = pose(fusionfilt);

    % Log the expected orientation and position
    expQ(ii) = q;
```

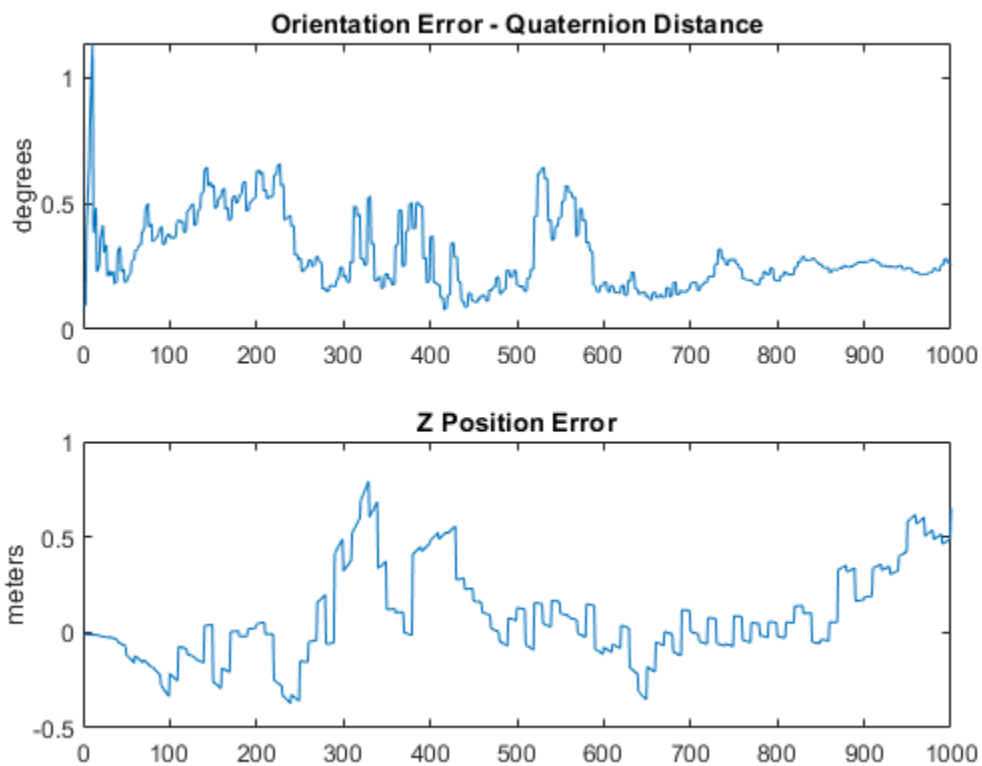
```
expP(ii) = pos(3);  
  
if usePoseViewer  
    hfunc.view(actP(ii), actQ(ii),expP(ii), expQ(ii)); %#ok<*UNRCH>  
end
```

end

Plot Filter Performance

Plot the performance of the filter. The display shows the error in the orientation using quaternion distance and height error.

```
hfunc.plotErrs(actP, actQ, expP, expQ);
```



Conclusion

This example shows how to use the `ahrs10filter` to perform 10-axis sensor fusion for height and orientation.

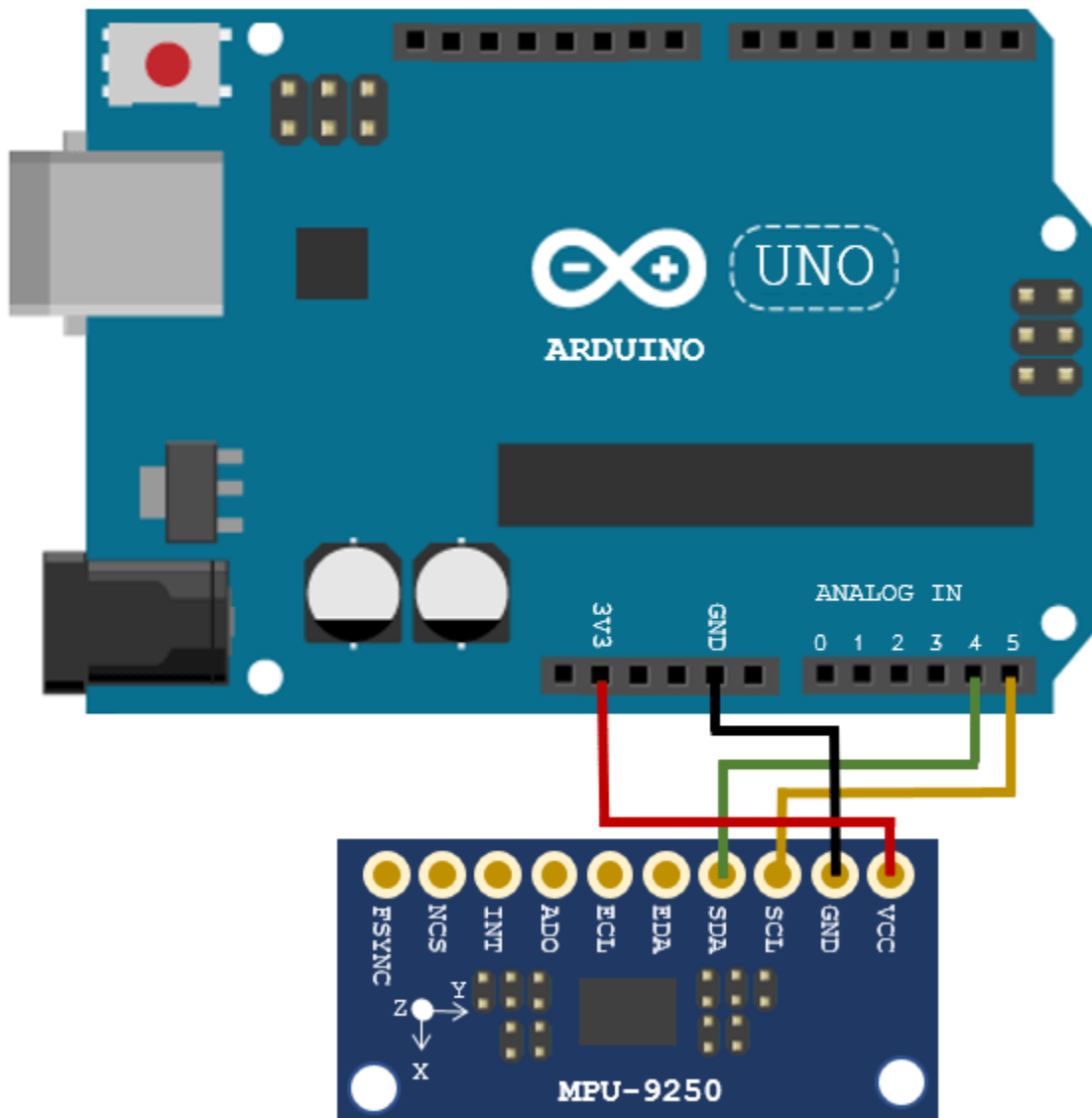
Estimate Orientation with a Complementary Filter and IMU Data

This example shows how to stream IMU data from an Arduino and estimate orientation using a complementary filter.

Connect Hardware

Connect the SDA, SCL, GND, and VCC pins of the MPU-9250 sensor to the corresponding pins of the Arduino® hardware. This example uses an Arduino® Uno board with the following connections:

- SDA - A4
- SCL - A5
- VCC - +3.3V
- GND - GND



Ensure that the connections to the sensors are intact. It is recommended to attach/connect the sensor to a prototype shield to avoid loose connections while the sensor is in motion. Refer the [Troubleshooting Sensors](#) page to debug the sensor related issues.

Create Sensor Object

Create an `arduino` object and an `mpu9250` object. Specify the sensor sampling rate F_s and the amount of time to run the loops. Optionally, enable the `isVerbose` flag to check if any samples are overrun. By disabling the `useHW` flag, you can also run the example with sensor data saved in the MAT-file `loggedMPU9250Data.mat`.

The data in `loggedMPU9250Data.mat` was logged while the IMU was generally facing due South, then rotated:

- +90 degrees around the z-axis

- -180 degrees around the z-axis
- +90 degrees around the z-axis
- +90 degrees around the y-axis
- -180 degrees around the y-axis
- +90 degrees around the y-axis
- +90 degrees around the x-axis
- -270 degrees around the x-axis
- +180 degrees around the x-axis

Notice that the last two rotations around the x-axis are an additional 90 degrees. This was done to flip the device upside-down. The final orientation of the IMU is the same as the initial orientation, due South.

```

Fs = 100;
samplesPerRead = 10;
runTime = 20;
isVerbose = false;
useHW = true;

if useHW
    a = arduino;
    imu = mpu9250(a, 'SampleRate', Fs, 'OutputFormat', 'matrix', ...
        'SamplesPerRead', samplesPerRead);
else
    load('loggedMPU9250Data.mat', 'allAccel', 'allGyro', 'allMag', ...
        'allT', 'allOverrun', ...
        'numSamplesAccelGyro', 'numSamplesAccelGyroMag')
end

```

Align Axes of MPU-9250 Sensor with NED Coordinates

The axes of the accelerometer, gyroscope, and magnetometer in the MPU-9250 are not aligned with each other. Specify the index and sign x-, y-, and z-axis of each sensor so that the sensor is aligned with the North-East-Down (NED) coordinate system when it is at rest. In this example, the magnetometer axes are changed while the accelerometer and gyroscope axes remain fixed. For your own applications, change the following parameters as necessary.

```

% Accelerometer axes parameters.
accelXAxisIndex = 1;
accelXAxisSign = 1;
accelYAxisIndex = 2;
accelYAxisSign = 1;
accelZAxisIndex = 3;
accelZAxisSign = 1;

% Gyroscope axes parameters.
gyroXAxisIndex = 1;
gyroXAxisSign = 1;
gyroYAxisIndex = 2;
gyroYAxisSign = 1;
gyroZAxisIndex = 3;
gyroZAxisSign = 1;

% Magnetometer axes parameters.

```

```
magXAxisIndex = 2;
magXAxisSign = 1;
magYAxisIndex = 1;
magYAxisSign = 1;
magZAxisIndex = 3;
magZAxisSign = -1;

% Helper functions used to align sensor data axes.

alignAccelAxes = @(in) [accelXAxisSign, accelYAxisSign, accelZAxisSign] ...
    .* in(:, [accelXAxisIndex, accelYAxisIndex, accelZAxisIndex]);

alignGyroAxes = @(in) [gyroXAxisSign, gyroYAxisSign, gyroZAxisSign] ...
    .* in(:, [gyroXAxisIndex, gyroYAxisIndex, gyroZAxisIndex]);

alignMagAxes = @(in) [magXAxisSign, magYAxisSign, magZAxisSign] ...
    .* in(:, [magXAxisIndex, magYAxisIndex, magZAxisIndex]);
```

Perform Additional Sensor Calibration

If necessary, you may calibrate the magnetometer to compensate for magnetic distortions. For more details, see the Compensating for Hard Iron Distortions section of the “Estimating Orientation Using Inertial Sensor Fusion and MPU-9250” (Sensor Fusion and Tracking Toolbox) example.

Specify Complementary filter Parameters

The `complementaryFilter` has two tunable parameters. The `AccelerometerGain` parameter determines how much the accelerometer measurement is trusted over the gyroscope measurement. The `MagnetometerGain` parameter determines how much the magnetometer measurement is trusted over the gyroscope measurement.

```
compFilt = complementaryFilter('SampleRate', Fs)
```

```
compFilt =
```

```
complementaryFilter with properties:
```

```
    SampleRate: 100
 AccelerometerGain: 0.0100
 MagnetometerGain: 0.0100
    HasMagnetometer: 1
 OrientationFormat: 'quaternion'
```

Estimate Orientation with Accelerometer and Gyroscope

Set the `HasMagnetometer` property to `false` to disable the magnetometer measurement input. In this mode, the filter only takes accelerometer and gyroscope measurements as inputs. Also, the filter assumes the initial orientation of the IMU is aligned with the parent navigation frame. If the IMU is not aligned with the navigation frame initially, there will be a constant offset in the orientation estimation.

```
compFilt = complementaryFilter('HasMagnetometer', false);
```

```
tuner = HelperOrientationFilterTuner(compFilt);
```

```
if useHW
```

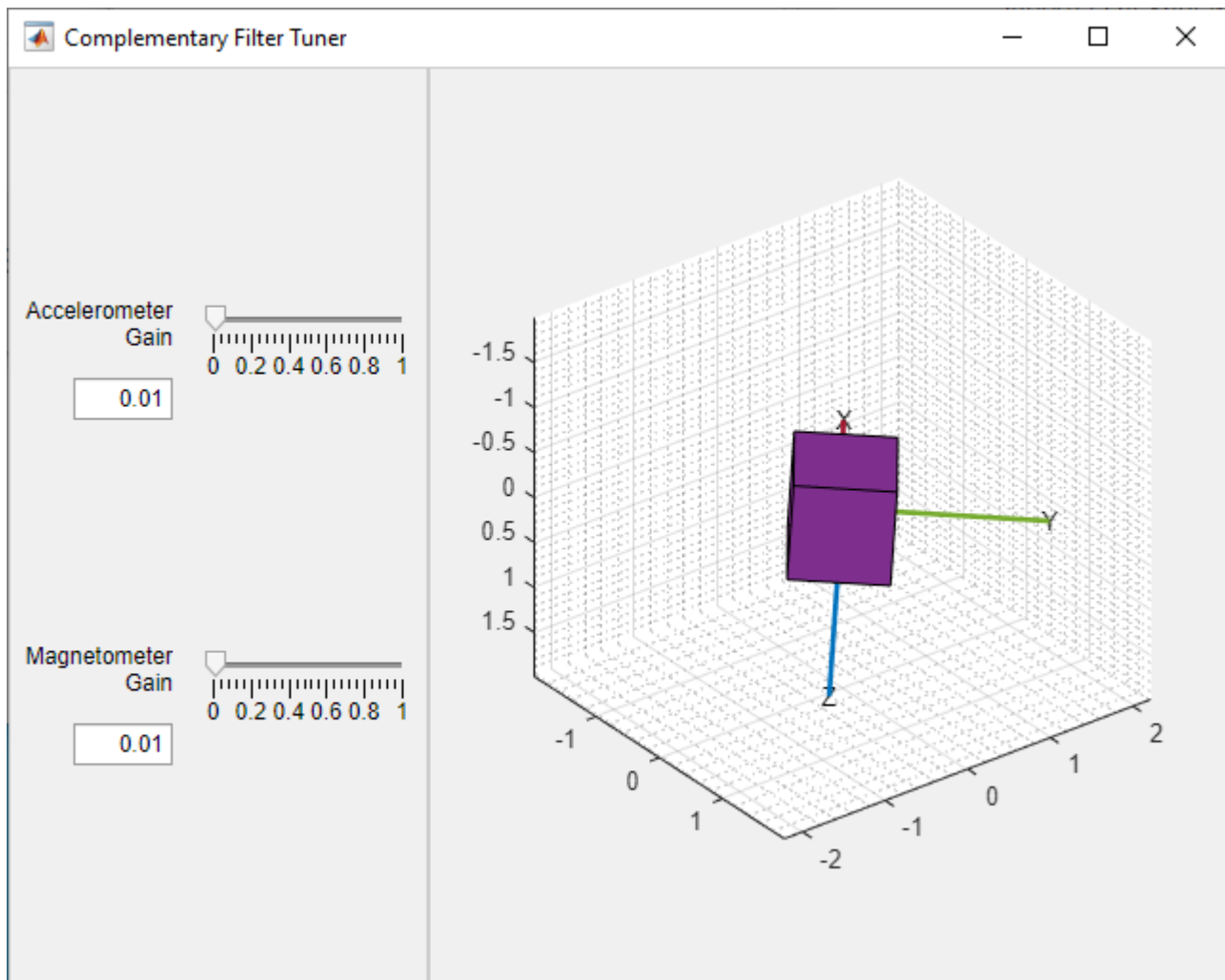
```
tic
else
    idx = 1:samplesPerRead;
    overrunIdx = 1;
end
while true
    if useHW
        [accel, gyro, mag, t, overrun] = imu();
        accel = alignAccelAxes(accel);
        gyro = alignGyroAxes(gyro);
    else
        accel = allAccel(idx,:);
        gyro = allGyro(idx,:);
        mag = allMag(idx,:);
        t = allT(idx,:);
        overrun = allOverrun(overrunIdx,:);

        idx = idx + samplesPerRead;
        overrunIdx = overrunIdx + 1;
        pause(samplesPerRead/Fs)
    end

    if (isVerbose && overrun > 0)
        fprintf('%d samples overrun ...\n', overrun);
    end

    q = compFilt(accel, gyro);
    update(tuner, q);

    if useHW
        if toc >= runTime
            break;
        end
    else
        if idx(end) > numSamplesAccelGyro
            break;
        end
    end
end
end
```



Estimate Orientation with Accelerometer, Gyroscope, and Magnetometer

With the default values of `AccelerometerGain` and `MagnetometerGain`, the filter trusts more on the gyroscope measurements in the short-term, but trusts more on the accelerometer and magnetometer measurements in the long-term. This allows the filter to be more reactive to quick orientation changes and prevents the orientation estimates from drifting over longer periods of time. For specific IMU sensors and application purposes, you may want to tune the parameters of the filter to improve the orientation estimation accuracy.

```
compFilt = complementaryFilter('SampleRate', Fs);
tuner = HelperOrientationFilterTuner(compFilt);

if useHW
    tic
end
while true
    if useHW
        [accel, gyro, mag, t, overrun] = imu();
        accel = alignAccelAxes(accel);
        gyro = alignGyroAxes(gyro);
```



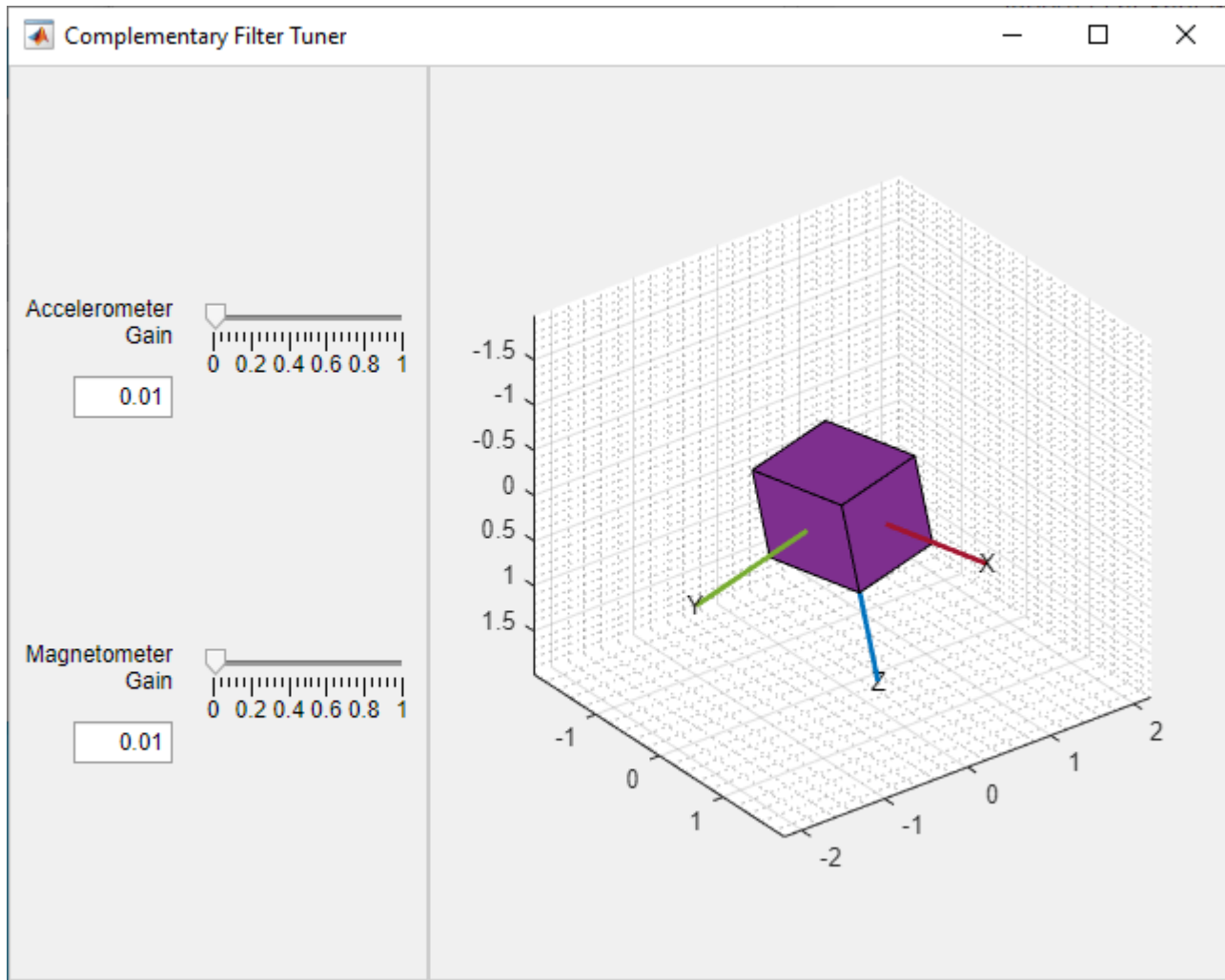
```
        mag = alignMagAxes(mag);
    else
        accel = allAccel(idx,:);
        gyro = allGyro(idx,:);
        mag = allMag(idx,:);
        t = allT(idx,:);
        overrun = allOverrun(overrunIdx,:);

        idx = idx + samplesPerRead;
        overrunIdx = overrunIdx + 1;
        pause(samplesPerRead/Fs)
    end

    if (isVerbose && overrun > 0)
        fprintf('%d samples overrun ...\n', overrun);
    end

    q = compFilt(accel, gyro, mag);
    update(tuner, q);

    if useHW
        if toc >= runTime
            break;
        end
    else
        if idx(end) > numSamplesAccelGyroMag
            break;
        end
    end
end
end
```



Summary

This example showed how to estimate the orientation of an IMU using data from an Arduino and a complementary filter. This example also showed how to configure the IMU and discussed the effects of tuning the complementary filter parameters.

Estimate Orientation Using AHRS Filter and IMU Data in Simulink

This example shows how to stream IMU data from sensors connected to Arduino® board and estimate orientation using AHRS filter and IMU sensor.

Required MathWorks Products

- MATLAB®
- Simulink®
- Simulink support Package for Arduino Hardware
- Either Navigation Toolbox™ or Sensor Fusion and Tracking Toolbox™

Hardware Required

1. Any of the Arduino board given below:

- Arduino Leonardo
- Arduino Mega 2560
- Arduino Mega ADK
- Arduino Micro
- Arduino Nano 3.0
- Arduino Uno
- Arduino Due
- Arduino MKR1000
- Arduino MKR WIFI 1010
- Arduino MKR ZERO
- Arduino Nano 33 IoT
- Arduino Nano 33 BLE Sense

2. IMU sensor with accelerometer, gyroscope, and magnetometer. In this example, X-NUCLEO-IKS01A2 sensor expansion board is used. The LSM6DSL sensor on the expansion board is used to get acceleration and angular rate values. The LSM303AGR sensor on the expansion board is used to get magnetic field value. The sensor data can be read using I2C protocol.

Hardware Connection

X-NUCLEO-IKS01A2 shield comes with Arduino UNO connectors, which makes it easy to interface with UNO board. For other boards, connect the SDA, SCL, 3.3V, and GND pin of the Arduino board to the respective pins on the sensor shield.

Hardware Configuration in the model

The example uses two models, AnalyseIMUData.slx and EstimateOrientationUsingAHRSandIMU.slx. Both the models are preconfigured to work with Arduino UNO. If you are using a different Arduino board, change the hardware board by doing the following steps:

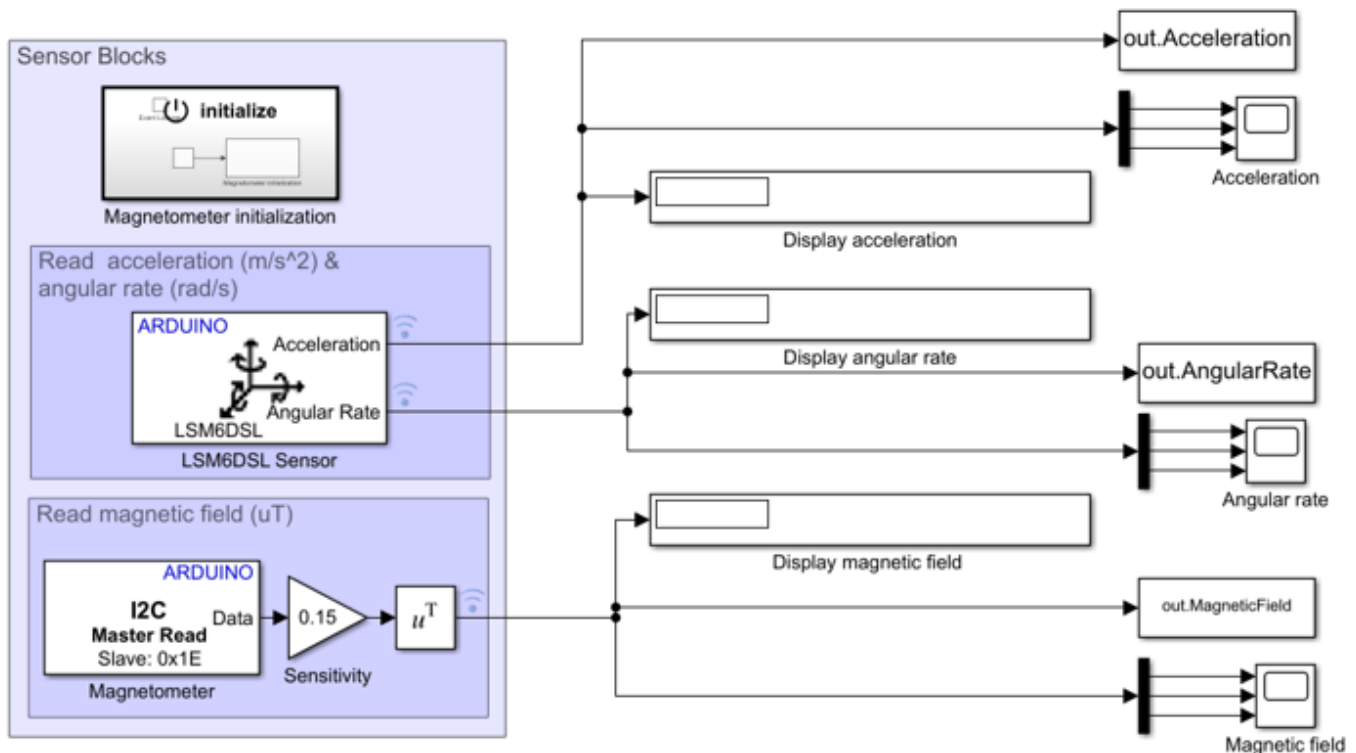
1. Click **Hardware Settings** in the **Hardware** tab of the Simulink toolbar.
2. In the Configurations Parameters dialog box, select **Hardware Implementation**.
3. From the **Hardware board** list, select the type of Arduino board that you are using.
4. Click **Apply**. Click **OK** to close the dialog box.

Task 1 - Read and Calibrate Sensor Values

This section describes how to read and calibrate the sensor values for the orientation estimation algorithm. To read and analyze values, use the model AnalyseIMUData.slx.

```
open_system('AnalyseIMUData.slx');
```

Analyse IMU Data



Copyright 2022 The MathWorks, Inc.

Reading acceleration and angular rate from LSM6DSL Sensor

Simulink Support Package for Arduino Hardware provides LSM6DSL IMU Sensor (Simulink Support Package for Arduino Hardware) block to read acceleration and angular rate along the X, Y and Z axis from LSM6DSL sensor connected to Arduino. The block outputs acceleration in m/s² and angular rate in rad/s. The sensor can be further configured by selecting the options given on the block mask.

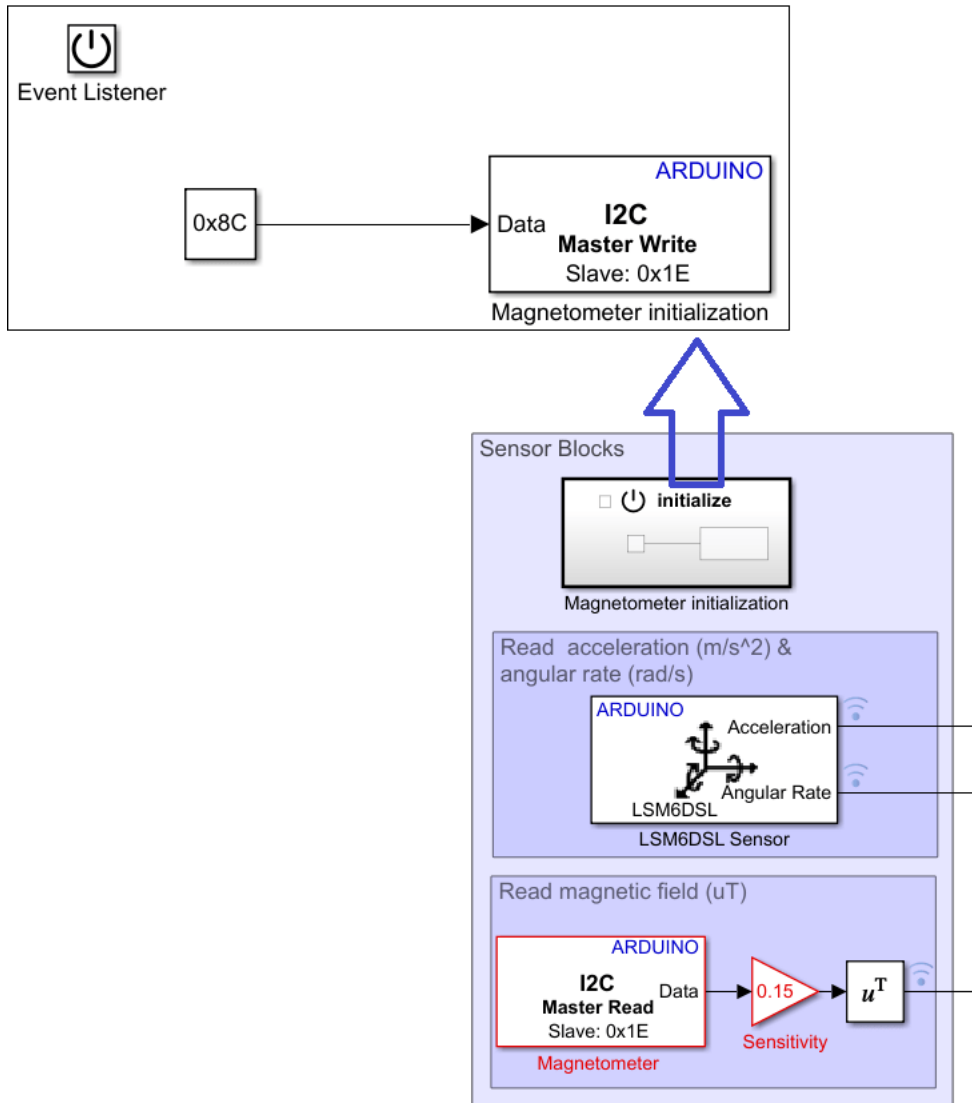
Reading sensor values form LSM303AGR

To read magnetic field values using LSM303AGR sensor, the example uses I2C Read (Simulink Support Package for Arduino Hardware) and I2C Write (Simulink Support Package for Arduino Hardware) blocks in the Support Package.

The I2C Address of the LSM303AGR sensor is 0x1E. This address is specified as *Slave address* in I2C Read/I2C Write block that is used to configure and read the magnetic field value from the sensor.

Depending on the Output Data Rate (ODR) required, a value needs to be written to CFG_REG_A_M register (0x60) of the sensor. To see the available ODRs, refer the LSM303AGR datasheet. This is a one-time operation required to initialize the sensor, and it is done using the Initialize block in the model.

The magnetic field is read from the output registers (0x68 - 0x6D) of the sensor using the I2C Read block and it is converted to microtesla as required by the example.



Perform Magnetometer calibration

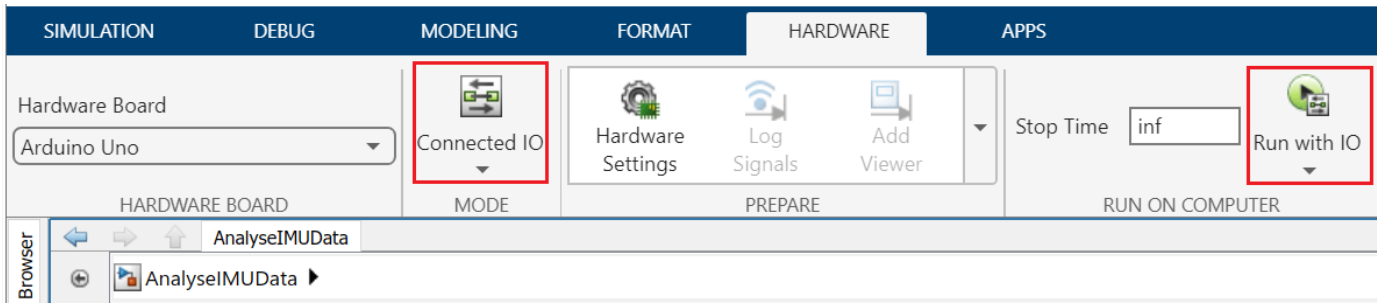
To get accurate measurements from the sensor, the sensor need to be calibrated. In this section, we consider magnetometer calibration for compensating hard iron distortions. Hard iron effects are stationary interfering magnetic noise sources. Often, these come from other metallic objects on the circuit board with the magnetometer. These distortions can be corrected by subtracting the correction value from the magnetometer readings for each axis.

To find the correction values, do the following:

1. Open the model *AnalyseIMUData*. The model uses the *To workspace* block (`out.MagneticField` in the model) to log magnetometer data.

The model is already configured to run in **Connected IO** mode. The simulation using Connected IO allows you to run your algorithm in Simulink with peripheral data from the hardware. For more details, refer to “Communicate with Hardware Using Connected IO” (Simulink Support Package for Arduino Hardware).

2. Run Connected IO by clicking the Run button corresponding to **Run with IO** under the **Hardware** tab.



3. While the model is running, rotate the sensor from 0 to 360 degree along each axis.

4. Click the Stop button to stop the Connected IO Simulation

6. The Magnetic field values are logged in the MATLAB base workspace as **out.MagneticField** variable. Use the magcal function on the logged values in MATLAB command window to obtain the correction coefficients.

```
[softIronFactor, hardIronOffset] = magcal(out.MagneticField);
```

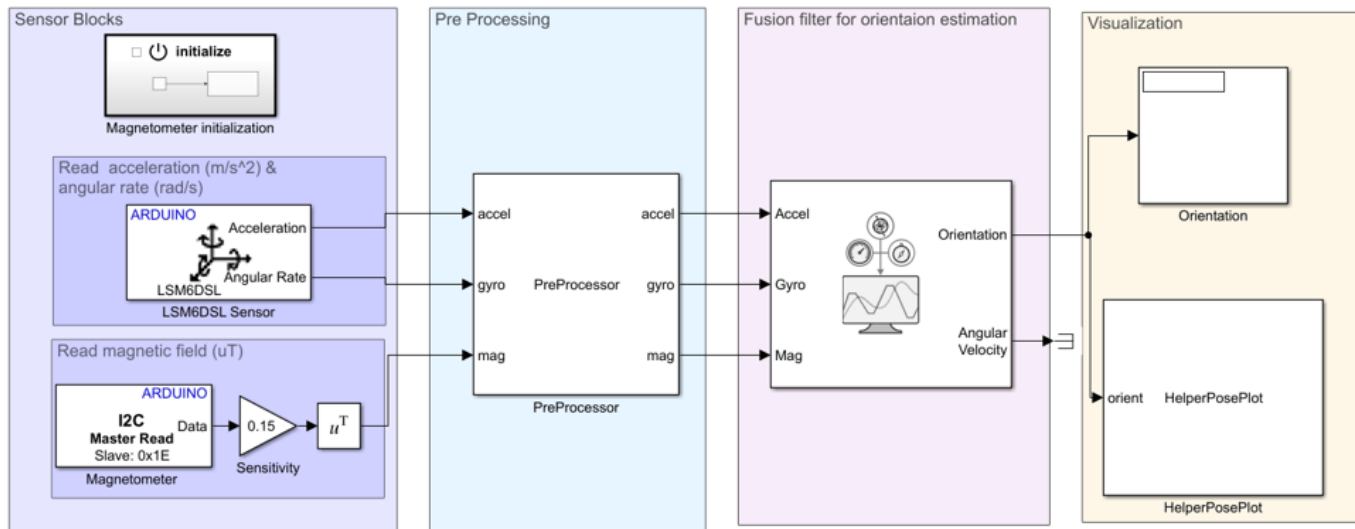
Note: The correction values change with the surroundings.

Task 2. Fuse Sensor Data with AHRS Filter

This section describes how to fuse the sensor data to estimate the orientation. Use the model *EstimateOrientationUsingAHRSAndIMU* for this section.

```
open_system("EstimateOrientationUsingAHRSandIMU.slx");
```

Estimate Orientation using AHRS filter and IMU sensor



Copyright 2022 The MathWorks, Inc.

Sensor Blocks

The first part of the model is for reading sensor values, which is described in the previous section. If you make changes to sensor blocks in the previous task, make the corresponding changes in the blocks in this model as well.

PreProcessor Block

The Preprocessor block in the model accepts acceleration, angular rate, and magnetic field from the sensor and magnetic field correction values. The block outputs the calibrated and axis-aligned sensor values.

Modify the values in the Constant block *Magnetometer correction values*, which is the input to the Preprocessor block, with the correction values (`hardIronOffset`) obtained from the step 6 in **Perform Magnetometer Calibration** section.

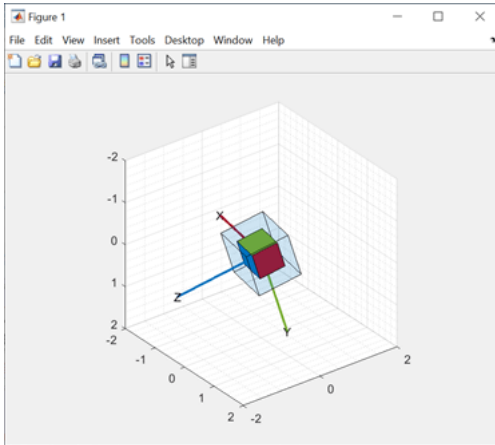
The axes of the accelerometer, gyroscope, and magnetometer in the sensor may not be aligned with each other. Specify the index and sign of x-, y-, and z-axis of each sensor on the PreProcessor block mask, so that the sensor is aligned with the North-East-Down (NED) coordinate system when it is at rest. In this example, the magnetometer Y-axes is changed while the accelerometer and gyroscope axes remain fixed.

Filter Block

To estimate orientation with IMU sensor data, an AHRS block is used. The AHRS block fuses accelerometer, magnetometer, and gyroscope sensor data to estimate device orientation. The AHRS block has tunable parameters. Tuning the parameters based on the specified sensors being used can improve performance. For more details, refer to [Tuning Filter Parameters](#) section in [Estimate Orientation Through Inertial Sensor Fusion](#).

Visualization Block

To visualize the orientation in Simulink, this example provides a helper block, *HelperPosePlot*. The block plots the pose specified by the quaternion or rotation matrix.

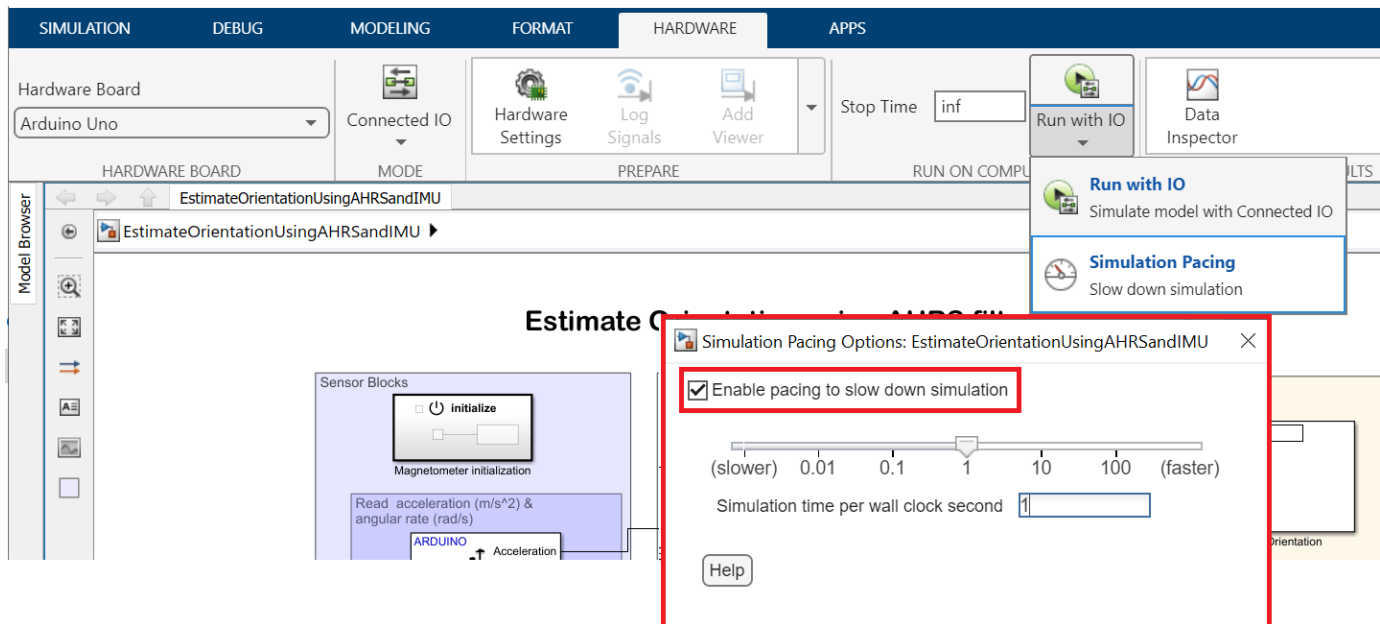


Validate the Model Design Using Connected IO

You can simulate the model in Connected IO to validate the model design before generating the code and deploying the model on Arduino board. This communication between the model and Arduino does not require any code generation or model deployment, thus accelerating the simulation process. For more information on Connected IO, see “Communicate with Hardware Using Connected IO” (Simulink Support Package for Arduino Hardware). The model is already configured to run in Connected IO mode.

This application requires the sensor data to be acquired in real time. To get real time data with Connected IO, you need to enable pacing. To acquire real time data from hardware, do the following:

1. On the Simulink toolbar, click the **Simulation** tab and set the Simulation mode to **Normal**.
2. To run this model in the Connected IO mode, click the **Hardware** tab, go to the **Mode** section, and select **Connected IO**.
3. On the **Hardware** tab, open the dropdown **Run with IO** in the **Run on Computer** section, and select **Simulation Pacing**.
4. Select **Enable pacing to slow down simulation**.



5. Click the Run icon corresponding to **Run with IO** to start the Connected IO Simulation.

Move the sensor and check if the motion in the figure is matching the motion of the sensor.

6. To stop running the model, click the Stop icon corresponding to **Run with IO**.

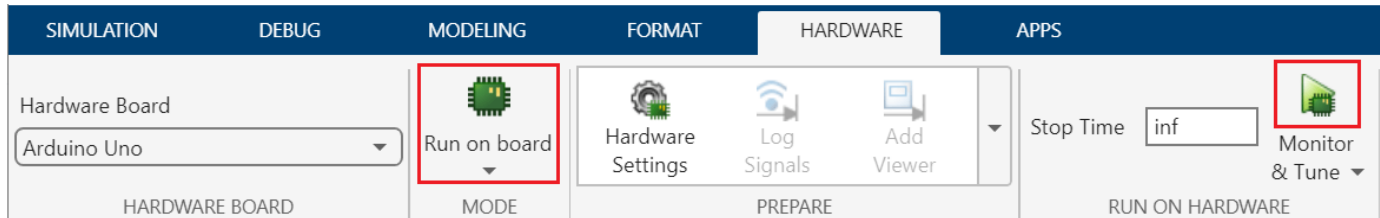
Run the Model in External Mode

After you successfully simulate the model in Connected IO, simulate the model in External mode. Unlike Connected IO, the model is deployed as a C code on the hardware. The code obtains real-time data from the hardware. In external mode, the data acquisition and parameter tuning are done while the application is running on the hardware.

Note: Ensure that the Arduino board you are using has sufficient memory to run the application on hardware. The boards like Arduino Uno, which have low memory, cannot support this application.

Note: The block *HelperPosePlot* is not supported for external mode workflow. To view the orientation in external mode, use other dashboards in Simulink like Scope, Display blocks, and so on.

1. To run External Mode, click the **Hardware** tab, go to the Mode section, select **Run on board (External mode)** and then click **Monitor & Tune**.



The lower left corner of the model window displays status while Simulink prepares, downloads, and runs the model on the hardware.

Move the sensor and verify the orientation values.

2. To stop running the model, click Stop corresponding to **Monitor and Tune**.

Estimate Phone Orientation Using Sensor Fusion

MATLAB Mobile™ reports sensor data from the accelerometer, gyroscope, and magnetometer on Apple or Android mobile devices. Raw data from each sensor or fused orientation data can be obtained. This examples shows how to compare the fused orientation data from the phone with the orientation estimate from the `ahrsfilter` object.

Read the Accelerometer, Gyroscope, Magnetometer, and Euler Angles

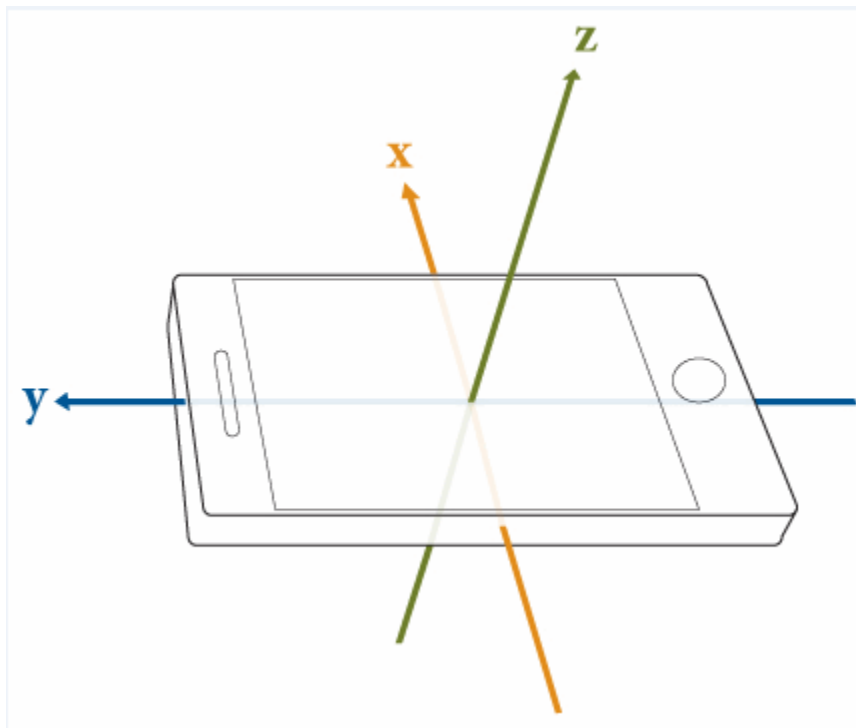
Read the logged phone sensor data. The MAT-file `samplePhoneData.mat` contains sensor data logged on an iPhone at a 100 Hz sampling rate. To run this example with your own phone data, refer to “Sensor Data Collection with MATLAB Mobile or MATLAB Online”.

```
matfile = 'samplePhoneData.mat';
SampleRate = 100; % This must match the data rate of the phone.
```

```
[Accelerometer, Gyroscope, Magnetometer, EulerAngles] ...
    = exampleHelperProcessPhoneData(matfile);
```

Convert to North-East-Down (NED) Coordinate Frame

MATLAB Mobile uses the convention shown in the following image. To process the sensor data with the `ahrsfilter` object, convert to NED, a right-handed coordinate system with clockwise motion around the axes corresponding to positive rotations. Swap the x- and y-axis and negate the z-axis for the various sensor data. Note that the accelerometer readings are negated since the readings have the opposite sign in the two conventions.



```
Accelerometer = -[Accelerometer(:,2), Accelerometer(:,1), -Accelerometer(:,3)];
Gyroscope = [Gyroscope(:,2), Gyroscope(:,1), -Gyroscope(:,3)];
Magnetometer = [Magnetometer(:,2), Magnetometer(:,1), -Magnetometer(:,3)];
```

```
qTrue = quaternion([EulerAngles(:,3), -EulerAngles(:,2), EulerAngles(:,1)], ...
    'eulerd', 'ZYX', 'frame');
```

Correct Phone Initial Rotation

The phone may have a random rotational offset. Without knowing the offset, you cannot compare the `ahrsfilter` object and the phone results. Use the first four samples to determine the rotational offset, then rotate the phone data back to desirable values.

```
% Get a starting guess at orientation using ecompass. No coefficients
% required. Use the initial orientation estimates to figure out what the
% phone's rotational offset is.
q = ecompass(Accelerometer, Magnetometer);

Navg = 4;
qfix = meanrot(q(1:Navg))./meanrot(qTrue(1:Navg));
Orientation = qfix*qTrue; % Rotationally corrected phone data.
```

Tune the AHRS Filter

To optimize the noise parameters for the phone, tune the `ahrsfilter` object. The parameters on the filter need to be tuned for the specific IMU on the phone that logged the data in the MAT-file. Use the `tune` function with the logged orientation data as ground truth.

```
orientFilt = ahrsfilter('SampleRate', SampleRate);
groundTruth = table(Orientation);
sensorData = table(Accelerometer, Gyroscope, Magnetometer);

tc = tunerconfig('ahrsfilter', "MaxIterations", 30, ...
    'Objectivelimit', 0.001, 'Display', 'none');
tune(orientFilt, sensorData, groundTruth, tc);
```

Fuse Sensor Data With Filter

Estimate the device orientation using the tuned `ahrsfilter` object.

```
reset(orientFilt);
qEst = orientFilt(Accelerometer, Gyroscope, Magnetometer);
```

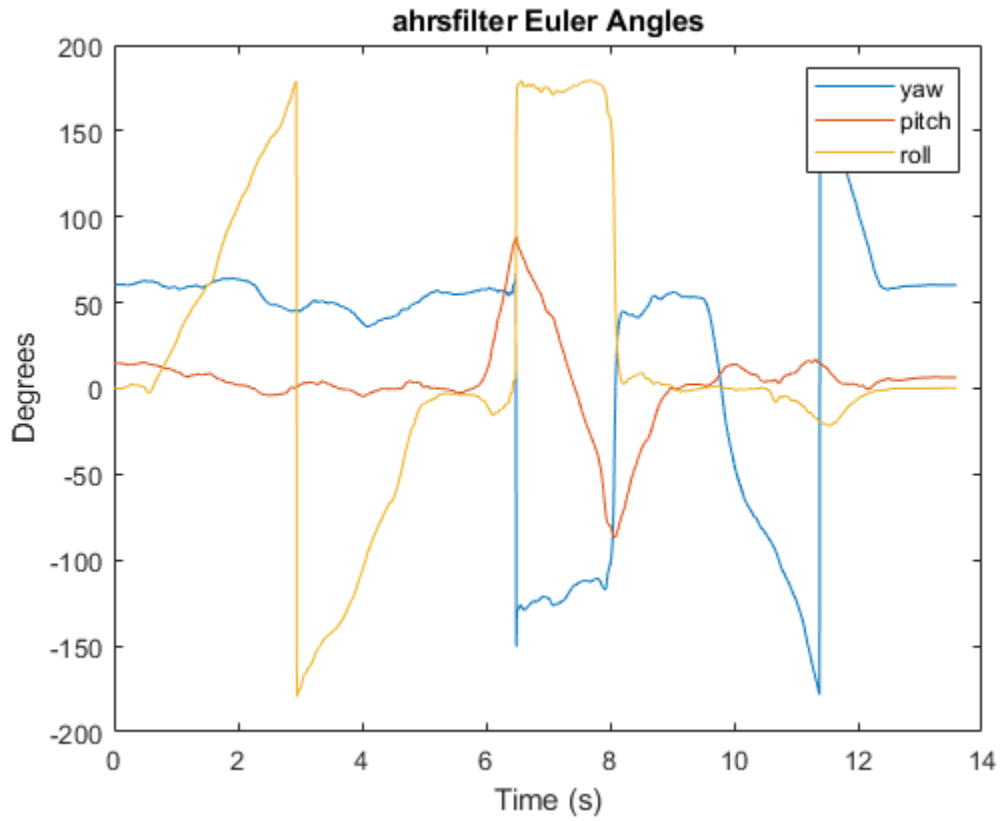
Plot Results

Plot the Euler angles for each orientation estimate and the quaternion distance between the two orientation estimates. Quaternion distance is measured as the angle between two quaternions. This distance can be used as an error metric for orientation estimation.

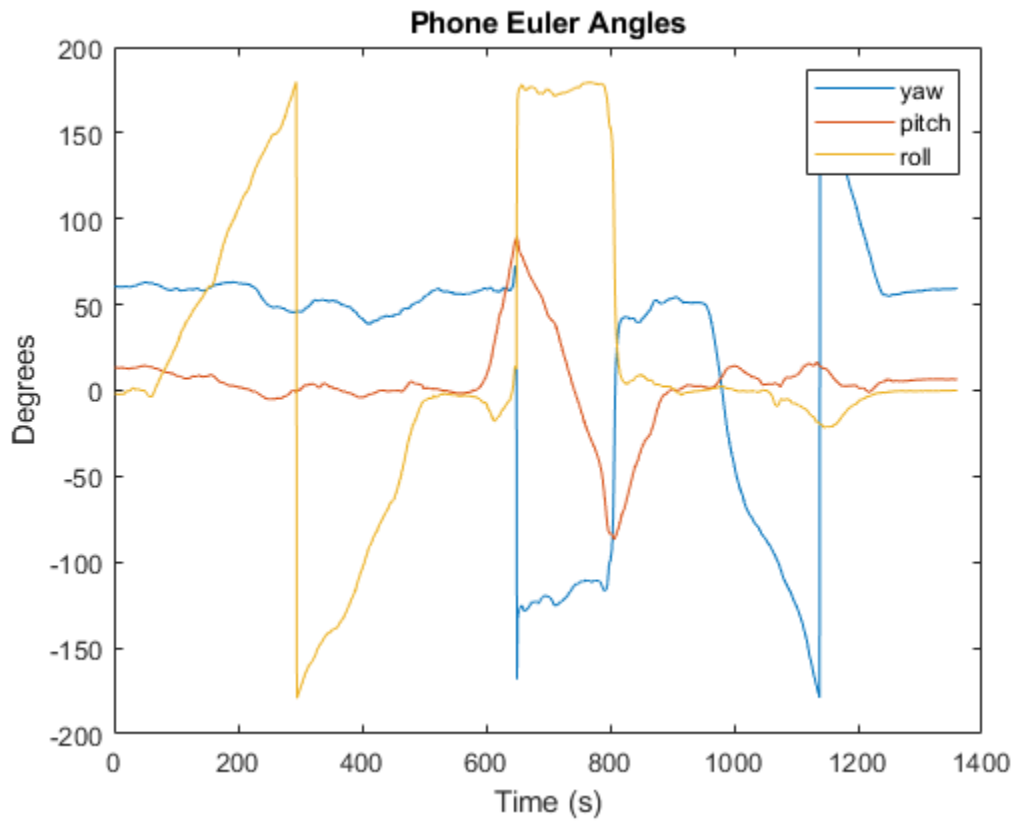
```
numSamples = numel(Orientation);
t = (0:numSamples-1).'/SampleRate;

d = rad2deg(dist(qEst, Orientation));

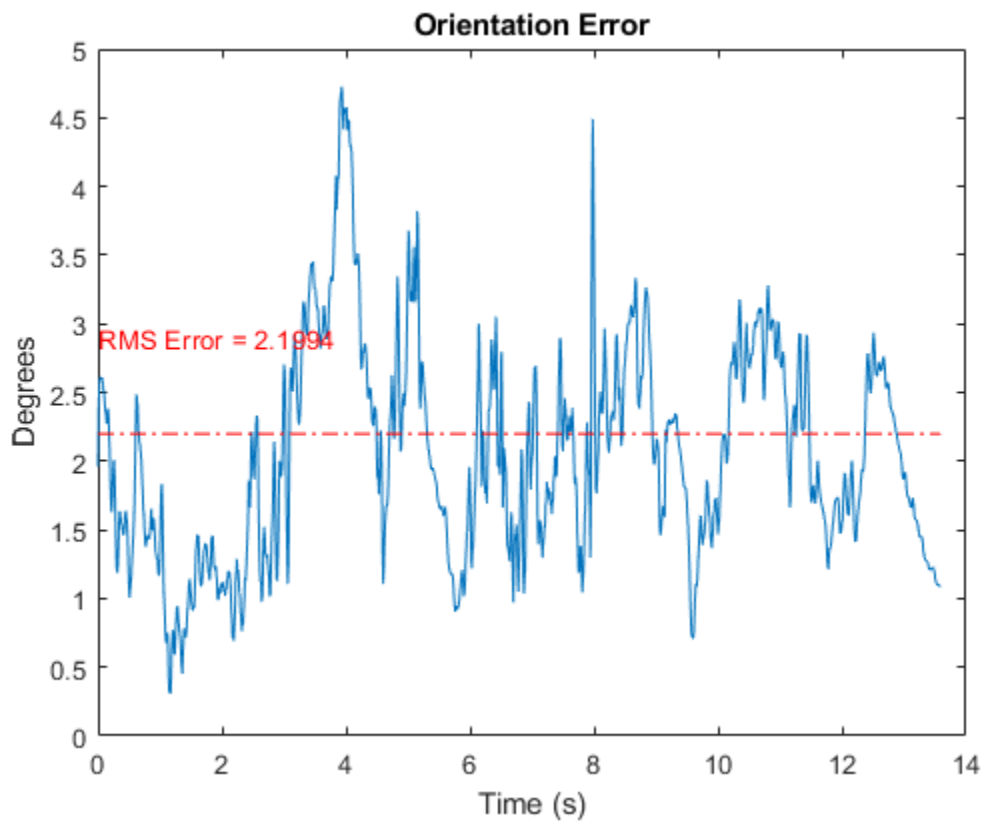
figure
plot(t, eulerd(qEst, 'ZYX', 'frame'))
legend yaw pitch roll
title('ahrsfilter Euler Angles')
ylabel('Degrees')
xlabel('Time (s)')
```



```
figure
plot(eulerd(Orientation, 'ZYX', 'frame'))
legend yaw pitch roll
title('Phone Euler Angles')
ylabel('Degrees')
xlabel('Time (s)')
```



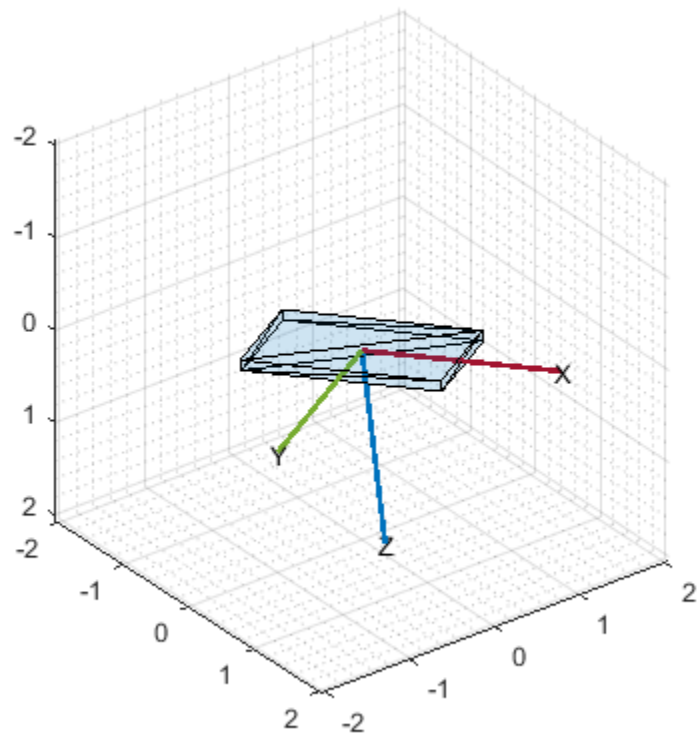
```
figure
plot(t, d)
title('Orientation Error')
ylabel('Degrees')
xlabel('Time (s)')
% Add RMS error
rmsval = sqrt(mean(d.^2));
line(t, repmat(rmsval,size(t)), 'LineStyle', '-.', 'Color', 'red');
text(t(1), rmsval + 0.7, "RMS Error = " + rmsval, 'Color', 'red')
```



Use `poseplot` to view the orientation estimates of the phone as a 3-D rectangle.

```
figure
pp = poseplot("MeshFileName", "phoneMesh.stl");

for i = 1:numel(qEst)
    set(pp, "Orientation", qEst(i));
    drawnow
end
```

Binaural Audio Rendering Using Head Tracking

Track head orientation by fusing data received from an IMU, and then control the direction of arrival of a sound source by applying head-related transfer functions (HRTF).

In a typical virtual reality setup, the IMU sensor is attached to the user's headphones or VR headset so that the perceived position of a sound source is relative to a visual cue independent of head movements. For example, if the sound is perceived as coming from the monitor, it remains that way even if the user turns his head to the side.

Required Hardware

- Arduino Uno
- Invensense MPU-9250

Hardware Connection

First, connect the Invensense MPU-9250 to the Arduino board. For more details, see Estimating Orientation Using Inertial Sensor Fusion and MPU-9250.

Create Sensor Object and IMU Filter

Create an arduino object.

```
a = arduino;
```

Create the Invensense MPU-9250 sensor object.

```
imu = mpu9250(a);
```

Create and set the sample rate of the Kalman filter.

```
Fs = imu.SampleRate;  
imufilt = imufilter('SampleRate',Fs);
```

Load the ARI HRTF Dataset

When sound travels from a point in space to your ears, you can localize it based on interaural time and level differences (ITD and ILD). These frequency-dependent ITD and ILD's can be measured and represented as a pair of impulse responses for any given source elevation and azimuth. The ARI HRTF Dataset contains 1550 pairs of impulse responses which span azimuths over 360 degrees and elevations from -30 to 80 degrees. You use these impulse responses to filter a sound source so that it is perceived as coming from a position determined by the sensor's orientation. If the sensor is attached to a device on a user's head, the sound is perceived as coming from one fixed place despite head movements.

First, load the HRTF dataset.

```
ARIDataset = load('ReferenceHRTF.mat');
```

Then, get the relevant HRTF data from the dataset and put it in a useful format for our processing.

```
hrtfData = double(ARIDataset.hrtfData);  
hrtfData = permute(hrtfData,[2,3,1]);
```

Get the associated source positions. Angles should be in the same range as the sensor. Convert the azimuths from [0,360] to [-180,180].

```
sourcePosition = ARIDataset.sourcePosition(:,[1,2]);
sourcePosition(:,1) = sourcePosition(:,1) - 180;
```

Load Monaural Recording

Load an ambisonic recording of a helicopter. Keep only the first channel, which corresponds to an omnidirectional recording. Resample it to 48 kHz for compatibility with the HRTF data set.

```
[heli,originalSampleRate] = audioread('Heli_16ch_ACN_SN3D.wav');
heli = 12*heli(:,1); % keep only one channel
```

```
sampleRate = 48e3;
heli = resample(heli,sampleRate,originalSampleRate);
```

Load the audio data into a `SignalSource` object. Set the `SamplesPerFrame` to 0.1 seconds.

```
sigsrc = dsp.SignalSource(heli, ...
    'SamplesPerFrame',sampleRate/10, ...
    'SignalEndAction','Cyclic repetition');
```

Set Up the Audio Device

Create an `audioDeviceWriter` with the same sample rate as the audio signal.

```
deviceWriter = audioDeviceWriter('SampleRate',sampleRate);
```

Create FIR Filters for the HRTF coefficients

Create a pair of FIR filters to perform binaural HRTF filtering.

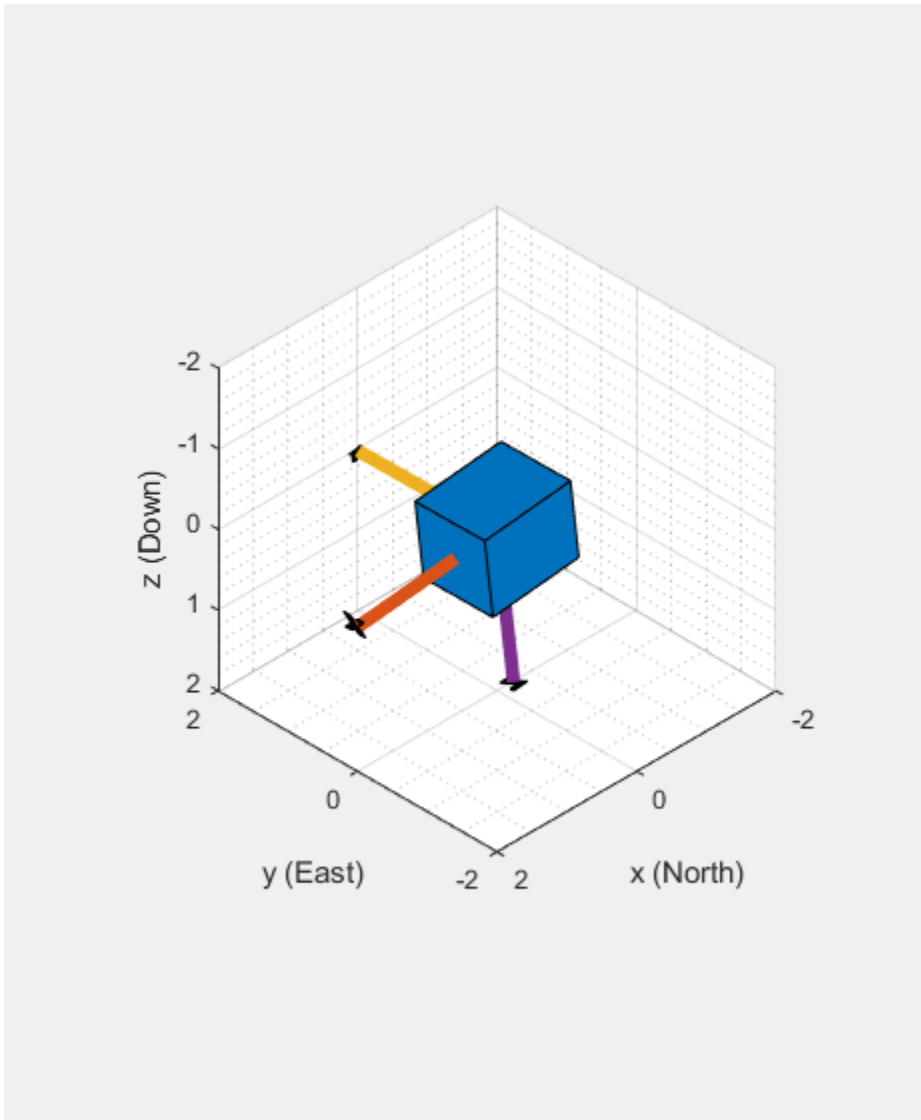
```
FIR = cell(1,2);
FIR{1} = dsp.FIRFilter('NumeratorSource','Input port');
FIR{2} = dsp.FIRFilter('NumeratorSource','Input port');
```

Initialize the Orientation Viewer

Create an object to perform real-time visualization for the orientation of the IMU sensor. Call the IMU filter once and display the initial orientation.

```
orientationScope = HelperOrientationViewer;
data = read(imu);

qimu = imufilt(data.Acceleration,data.AngularVelocity);
orientationScope(qimu);
```



Audio Processing Loop

Execute the processing loop for 30 seconds. This loop performs the following steps:

- 1 Read data from the IMU sensor.
- 2 Fuse IMU sensor data to estimate the orientation of the sensor. Visualize the current orientation.
- 3 Convert the orientation from a quaternion representation to pitch and yaw in Euler angles.
- 4 Use `interpolateHRTF` to obtain a pair of HRTFs at the desired position.
- 5 Read a frame of audio from the signal source.
- 6 Apply the HRTFs to the mono recording and play the stereo signal. This is best experienced using headphones.

```
imu0verruns = 0;  
audioUnderruns = 0;  
audioFiltered = zeros(sigsrc.SamplesPerFrame,2);
```

```
tic
while toc < 30

    % Read from the IMU sensor.
    [data,overrun] = read(imu);
    if overrun > 0
        imuOverruns = imuOverruns + overrun;
    end

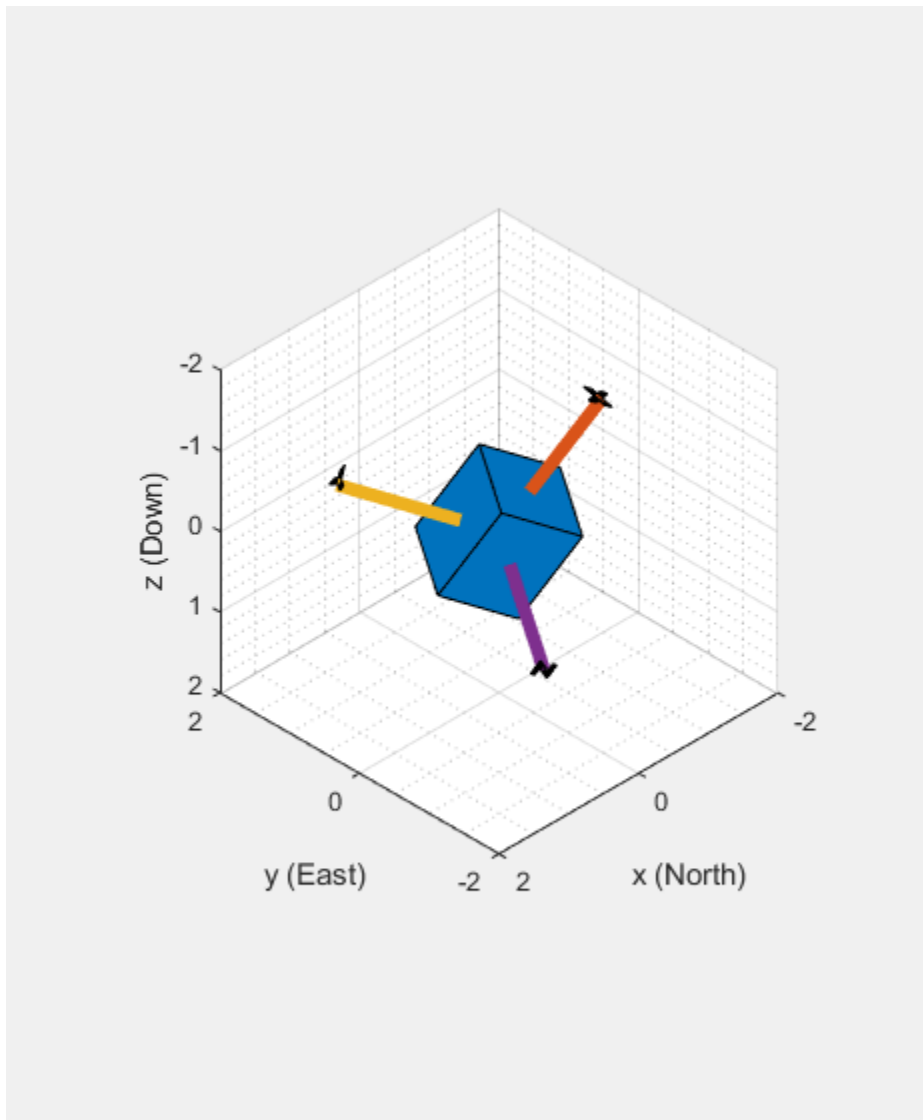
    % Fuse IMU sensor data to estimate the orientation of the sensor.
    qimu = imufilt(data.Acceleration,data.AngularVelocity);
    orientationScope(qimu);

    % Convert the orientation from a quaternion representation to pitch and yaw in Euler angles.
    ypr = eulerd(qimu,'zyx','frame');
    yaw = ypr(end,1);
    pitch = ypr(end,2);
    desiredPosition = [yaw,pitch];

    % Obtain a pair of HRTFs at the desired position.
    interpolatedIR = squeeze(interpolateHRTF(hrtfData,sourcePosition,desiredPosition));

    % Read audio from file
    audioIn = sigsrc();

    % Apply HRTFs
    audioFiltered(:,1) = FIR{1}(audioIn, interpolatedIR(1,:)); % Left
    audioFiltered(:,2) = FIR{2}(audioIn, interpolatedIR(2,:)); % Right
    audioUnderruns = audioUnderruns + deviceWriter(squeeze(audioFiltered));
end
```



Cleanup

Release resources, including the sound device.

```
release(sigsrc)  
release(deviceWriter)  
clear imu a
```

Estimating Orientation Using Inertial Sensor Fusion and MPU-9250

This example shows how to get data from an InvenSense MPU-9250 IMU sensor, and to use the 6-axis and 9-axis fusion algorithms in the sensor data to compute orientation of the device.

MPU-9250 is a 9-axis sensor with accelerometer, gyroscope, and magnetometer. The accelerometer measures acceleration, the gyroscope measures angular velocity, and the magnetometer measures magnetic field in x-, y- and z- axis. The axis of the sensor depends on the make of the sensor.

Required MathWorks® Products

- MATLAB®
- MATLAB Support Package for Arduino® Hardware
- Sensor Fusion and Tracking Toolbox™ or Navigation Toolbox™

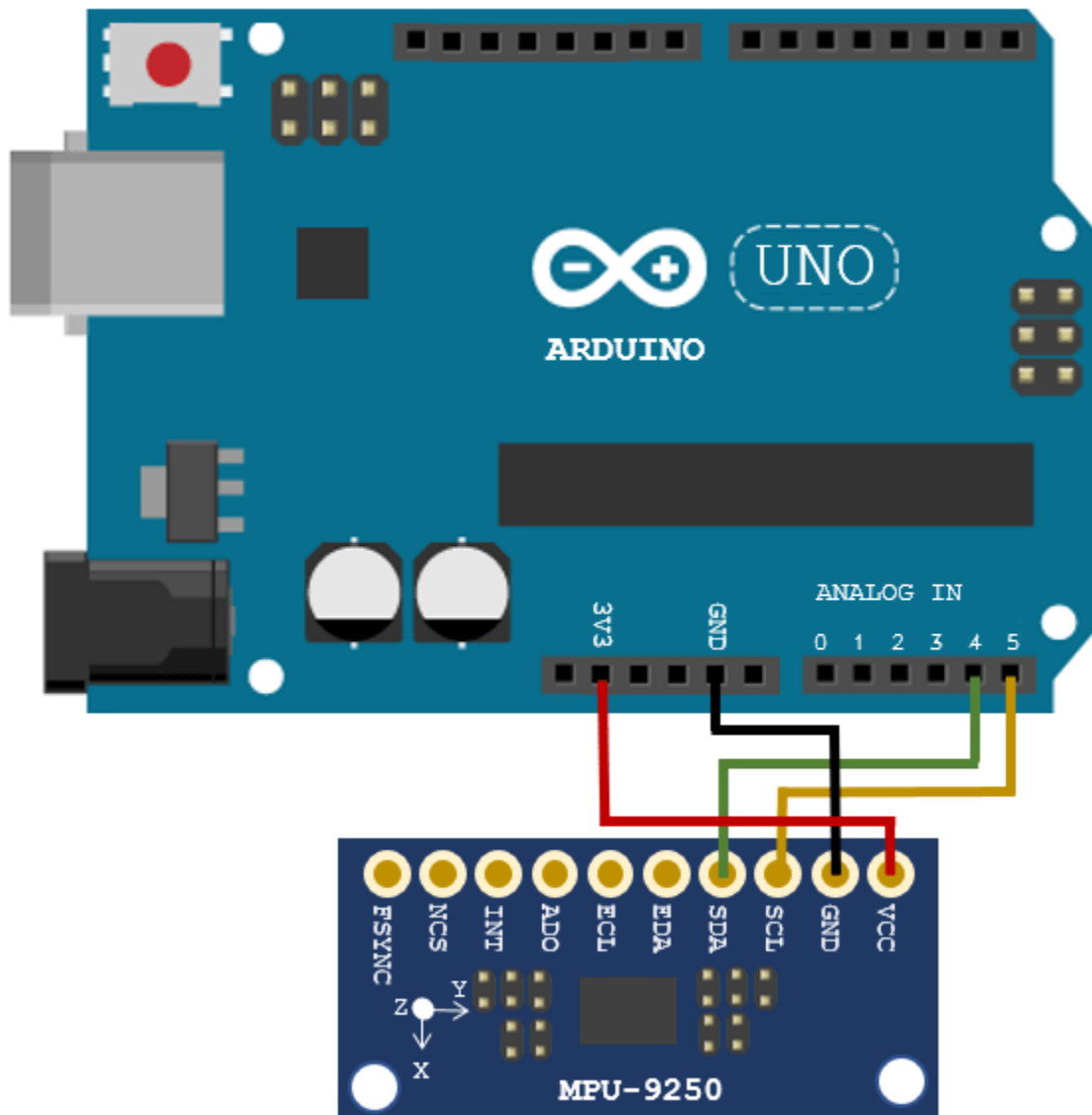
Required Hardware

- Arduino Uno
- InvenSense MPU-9250

Hardware Connection

Connect the SDA, SCL, GND, and the VCC pins of the MPU-9250 sensor to the corresponding pins on the Arduino® Hardware. This example uses the Arduino Uno board with the following connections:

- SDA - A4
- SCL - A5
- VCC - +3.3V
- GND - GND



Ensure that the connections to the sensors are intact. It is recommended to use a prototype shield and solder the sensor to it to avoid loose connections while moving the sensor. Refer the “Troubleshooting Sensors” (MATLAB Support Package for Arduino Hardware) page for sensors to debug the sensor related issues.

Create Sensor Object

Create an arduino object and include the I2C library.

```
a = arduino('COM9', 'Uno', 'Libraries', 'I2C');
```

Updating server code on board Uno (COM9). This may take a few minutes.

Create the MPU-9250 sensor object.

```
fs = 100; % Sample Rate in Hz
imu = mpu9250(a, 'SampleRate', fs, 'OutputFormat', 'matrix');
```


Compensating For Hard Iron Distortions

Fusion algorithms use magnetometer readings which need to compensate for magnetic distortions such as hard iron distortion. Hard iron distortions are produced by materials which create a magnetic field, resulting in shifting the origin on the response surface. These distortions can be corrected by subtracting the correction values from the magnetometer readings for each axis. In order to find the correction values, do the following:

1. Rotate the sensor from 0 to 360 degree around each axis.
2. Obtain the minimum and maximum magnetometer readings.
3. Average the minimum and maximum readings to get the correction values for each axis.

The correction values change with the surroundings.

The following code snippets can be used to obtain bias values for x axis, similar procedure can be followed for other axes as well:

```
displayMessage(['Fusion algorithms use magnetometer readings which need to compensate for magnetometer readings. The given code snippet can be used to find the correction values for compensating Hard Iron Distortions. This code is executing, rotate the sensor around x axis from 0 to 360 degree. For other axes, modify the code to rotate the sensor along that axis'],'Compensating Hard Iron Distortions');
tic;
stopTimer = 100;
magReadings=[];
while(toc<stopTimer)
    % Rotate the sensor around x axis from 0 to 360 degree.
    % Take 2-3 rotations to improve accuracy.
    % For other axes, rotate around that axis.
    [accel,gyro,mag] = read(imu);
    magReadings = [magReadings;mag];
end

% For y axis, use magReadings(:,2) and for z axis use magReadings(:,3)
magx_min = min(magReadings(:,1));
magx_max = max(magReadings(:,1));
magx_correction = (magx_max+magx_min)/2;
```

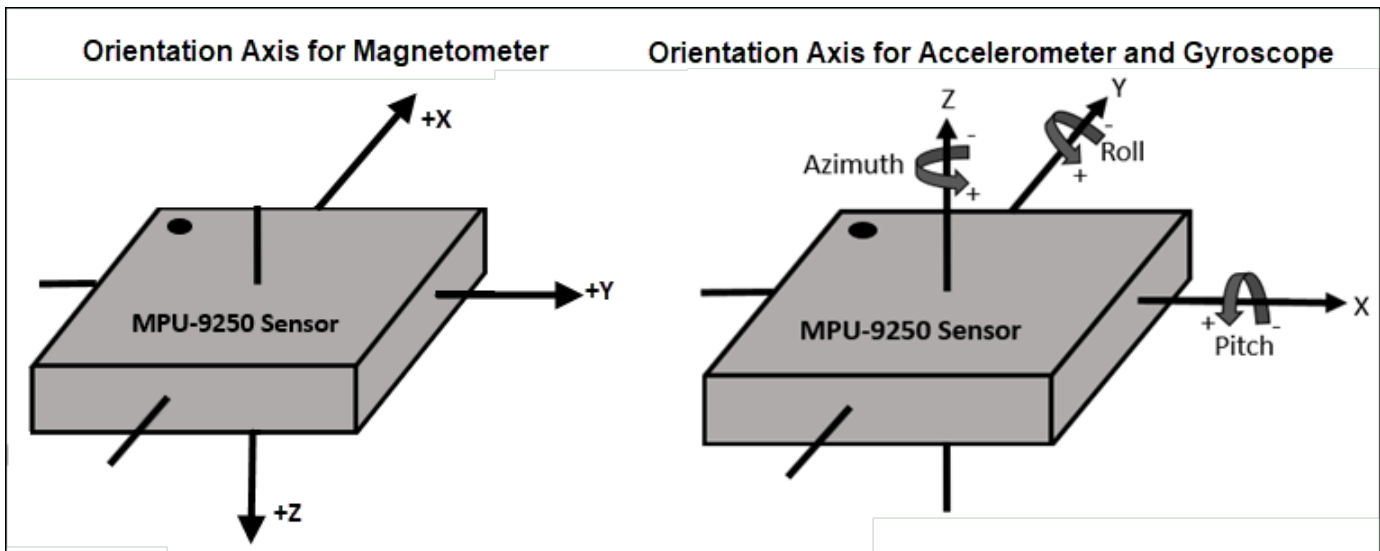
For more accurate tracking, calibrate the magnetometer for other distortions as well. The `magcal` function (this function is available in both the Sensor Fusion and Tracking Toolbox™ and the Navigation Toolbox™) can be used to compensate soft iron distortions as well. Change the correction values calculated for your sensor in the `readSensorDataMPU9250` function in the example folder.

Aligning the axis of MPU-9250 sensor with NED Coordinates

Sensor fusion algorithms used in this example use North-East-Down(NED) as a fixed, parent coordinate system. In the NED reference frame, the X-axis points north, the Y-axis points east, and the Z-axis points down. Depending on the algorithm, north may either be the magnetic north or true north. The algorithms in this example use the magnetic north. The algorithms used here expects all the sensors in the object to have their axis aligned and is in accordance with NED convention.

MPU-9250 has two devices, the magnetometer and the accelerometer-gyroscope, on the same board. The axes of these devices are different from each other. The magnetometer axis is aligned with the NED coordinates. The axis of the accelerometer-gyroscope is different from magnetometer in MPU-9250. The accelerometer and the gyroscope axis need to be swapped and/or inverted to match

the magnetometer axis. For more information refer to the section “Orientation of Axes” section in MPU-9250 datasheet.



To align MPU-9250 accelerometer-gyroscope axes to NED coordinates, do the following:

1. Define device axes: Define the imaginary axis as the device axis on the sensor in accordance to NED coordinate system which may or may not be same as sensor axes. For MPU-9250, magnetometer axis can be considered as device axis.

2. Swap the x and y values of accelerometer and gyroscope readings, so that the accelerometer and gyroscope axis is aligned with magnetometer axis.

3. Determine polarity values for accelerometer, and gyroscope.

a. Accelerometer

- Place the sensor such that device X axis is pointing downwards, perpendicular to the surface at which sensor is kept. Accelerometer readings should read approximately [9.8 0 0]. If not negate the x-values of accelerometer.
- Place the sensor such that device Y axis is pointing downwards, perpendicular to the surface at which sensor is kept. Accelerometer readings should read approximately [0 9.8 0]. If not negate the y-values of accelerometer.
- Place the sensor such that device Z axis is pointing downwards, perpendicular to the surface at which sensor is kept. Accelerometer readings should read approximately [0 0 9.8]. If not negate the z-values of accelerometer.

b. Gyroscope

Rotate the sensor along each axis and capture the readings. Use the right hand screw rule to correct the polarity of rotation.

The above method is used to set the axis of the sensor in this example.

Tuning Filter Parameters

The algorithms used in this example, when properly tuned, enable estimation of orientation and are robust against environmental noise sources. You must consider the situations in which the sensors

are used and tune the filters accordingly. See “Custom Tuning of Fusion Filters” (Sensor Fusion and Tracking Toolbox) for more details related to tuning filter parameters.

The example demonstrates three algorithms to determine orientation, namely `ahrsfilter`, `imufilter`, and `ecompass`. Refer “Determine Orientation Using Inertial Sensors” (Sensor Fusion and Tracking Toolbox) for more details related to inertial fusion algorithms.

Accelerometer-Gyroscope-Magnetometer Fusion

An attitude and heading reference system (AHRS) consist of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation of the device. The `ahrsfilter` produces a smoothly changing estimate of orientation of the device, while correctly estimating the north direction. The `ahrsfilter` has the ability to remove gyroscope bias and can also detect and reject mild magnetic jamming.

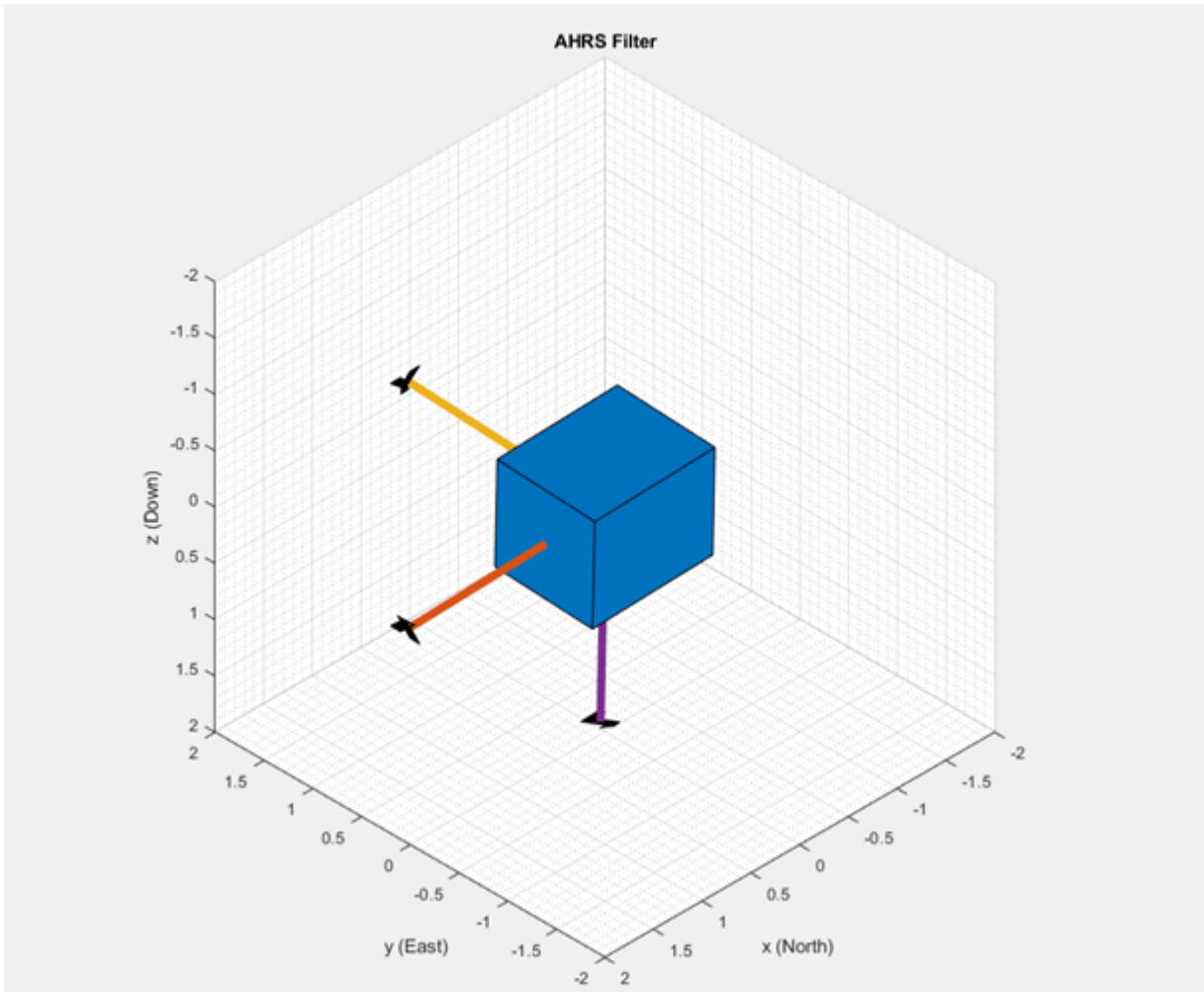
The following code snippets use `ahrsfilter` system object to determine orientation of the sensor and creates a figure which gets updated as you move the sensor. The sensor has to be stationary, before the start of this example.

```
% GyroscopeNoise and AccelerometerNoise is determined from datasheet.
GyroscopeNoiseMPU9250 = 3.0462e-06; % GyroscopeNoise (variance value) in units of rad/s
AccelerometerNoiseMPU9250 = 0.0061; % AccelerometerNoise(variance value)in units of m/s^2
viewer = HelperOrientationViewer('Title',{'AHRS Filter'});
FUSE = ahrsfilter('SampleRate',imu.SampleRate, 'GyroscopeNoise',GyroscopeNoiseMPU9250,'Accelerom
stopTimer = 100;
```

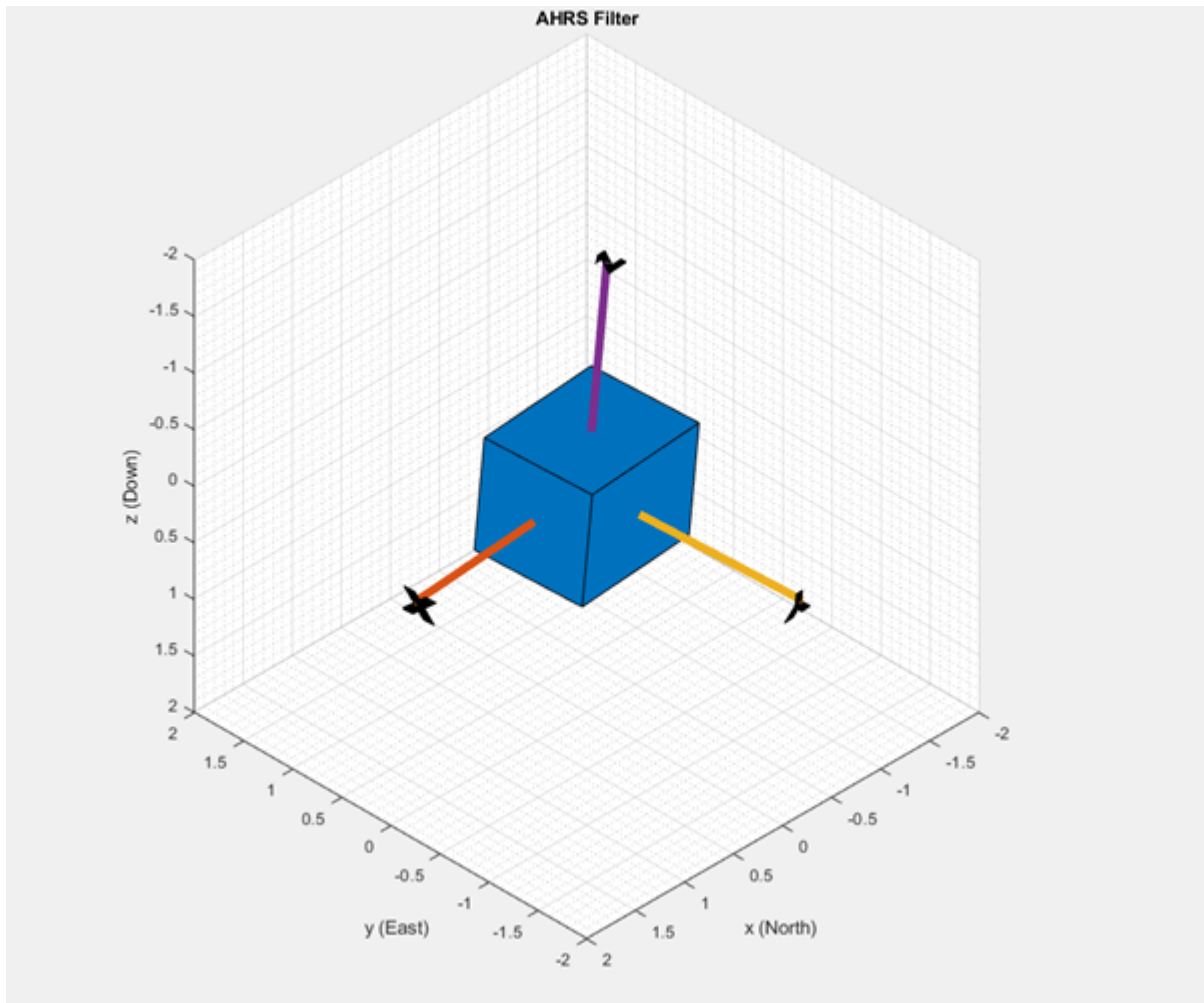
While the below code is getting executed, slowly move the sensor and check if the motion in the figure matches the motion of the sensor.

```
% Use ahrsfilter to estimate orientation and update the viewer as the
% sensor moves for time specified by stopTimer
displayMessage(['This section uses AHRS filter to determine orientation of the sensor by collect
    'system object. Move the sensor to visualize orientation of the sensor in the figure window.
    'click OK'],...
    'Estimate Orientation using AHRS filter and MPU-9250')
tic;
while(toc < stopTimer)
    [accel,gyro,mag] = readSensorDataMPU9250(imu);
    rotators = FUSE(accel,gyro,mag);
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

When the device X axis of sensor is pointing to north, the device Y-axis is pointing to east and device Z-axis is pointing down.



When the device X axis of sensor is pointing to north, device Y-axis is pointing to west and device Z-axis is pointing upwards.



Accelerometer-Gyroscope Fusion

The `imufilter` system object fuses accelerometer and gyroscope data using an internal error-state Kalman filter. The filter is capable of removing the gyroscope bias noise, which drifts over time. The filter does not process magnetometer data, so it does not correctly estimate the direction of north. The algorithm assumes the initial position of the sensor is in such a way that device X-axis of the sensor is pointing towards magnetic north, the device Y-axis of the sensor is pointing to east and the device Z-axis of the sensor is pointing downwards. The sensor must be stationary, before the start of this example.

The following code snippets use `imufilter` object to determine orientation of the sensor and creates a figure which gets updated as you move the sensor.

```
displayMessage(['This section uses IMU filter to determine orientation of the sensor by collect
'system object. Move the sensor to visualize orientation of the sensor in the figure window.
'click OK'],...
'Estimate Orientation using IMU filter and MPU-9250.')
```

```
% GyroscopeNoise and AccelerometerNoise is determined from datasheet.
GyroscopeNoiseMPU9250 = 3.0462e-06; % GyroscopeNoise (variance) in units of rad/s
AccelerometerNoiseMPU9250 = 0.0061; % AccelerometerNoise (variance) in units of m/s^2
viewer = HelperOrientationViewer('Title',{'IMU Filter'});
```

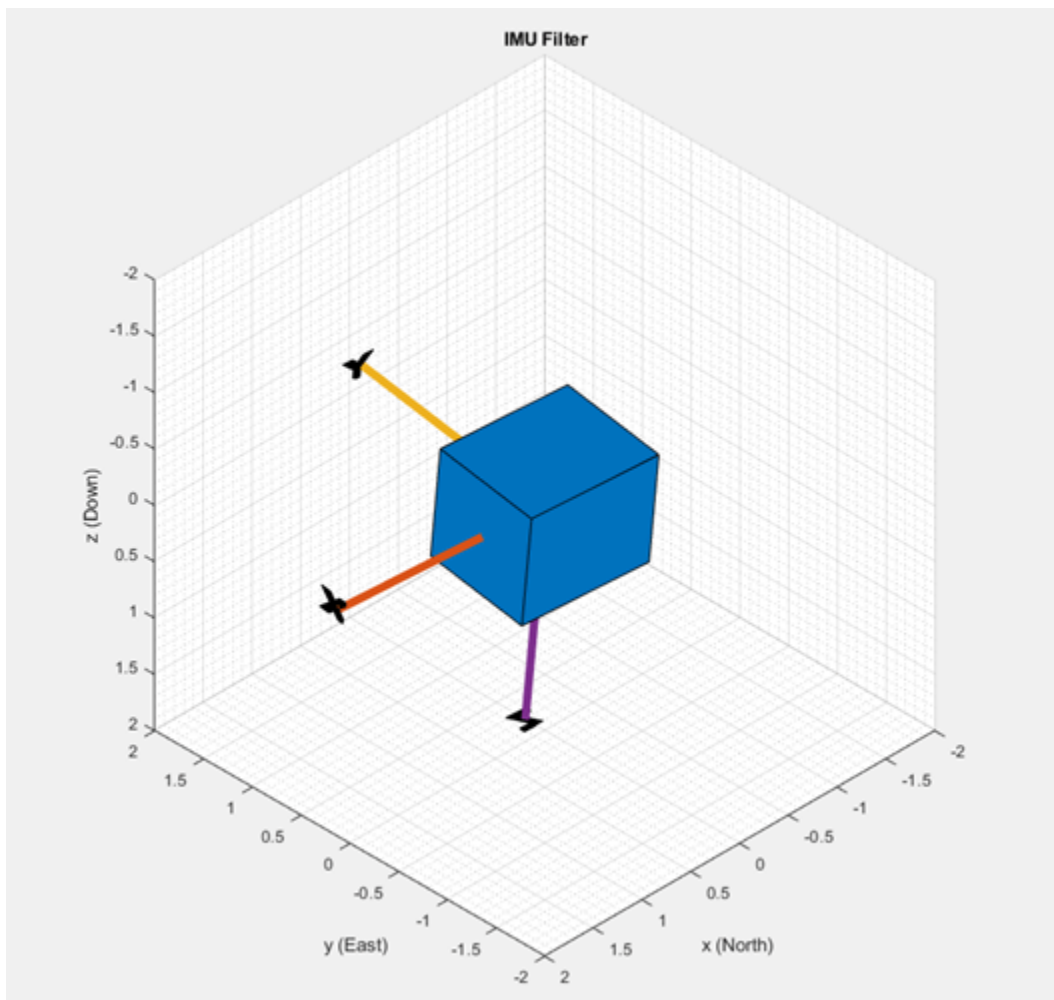
```
FUSE = imufilter('SampleRate',imu.SampleRate, 'GyroscopeNoise',GyroscopeNoiseMPU9250,'Accelerome
stopTimer=100;
```

While the below code is getting executed, slowly move the sensor and check if the motion in the figure matches the motion of the sensor.

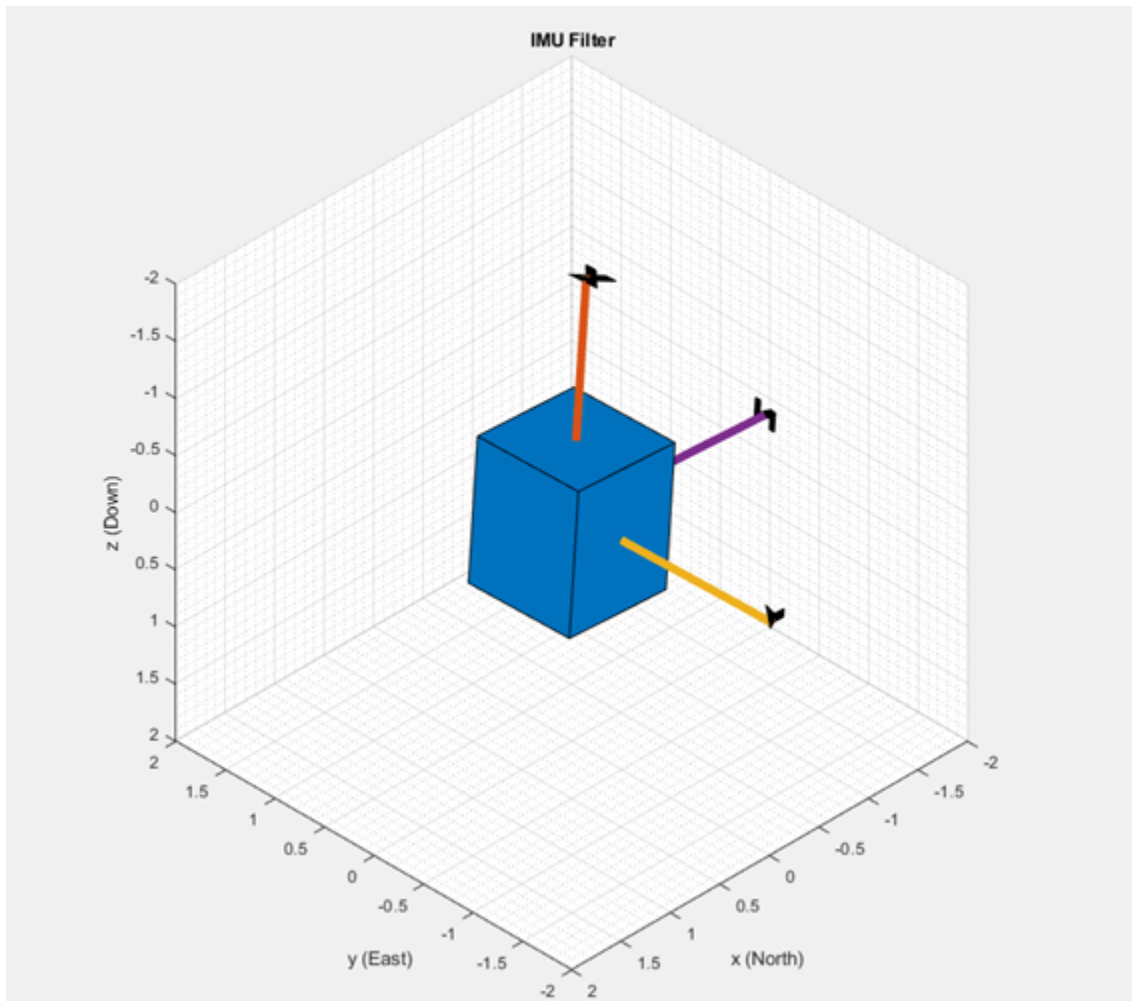
```
% Use imufilter to estimate orientation and update the viewer as the
% sensor moves for time specified by stopTimer
tic;
while(toc < stopTimer)
    [accel,gyro] = readSensorDataMPU9250(imu);
    rotators = FUSE(accel,gyro);
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

The `imufilter` algorithm can also be used with MPU6050 as well, since it does not require magnetometer values.

When the device X axis of sensor is pointing to north, device Z-axis is pointing downwards and device Y-axis is pointing to east.



When the device X axis of sensor is pointing upwards, device Y-axis is points to west and device Z-axis points to south.



Accelerometer-Magnetometer Fusion

The ecompass system object fuses the accelerometer and magnetometer data. The Ecompass algorithm is a memoryless algorithm that requires no parameter tuning but is highly susceptible to sensor noise. You could use spherical linear interpolation (SLERP) to lowpass filter a noisy trajectory. Refer “Lowpass Filter Orientation Using Quaternion SLERP” (Sensor Fusion and Tracking Toolbox) example for more details

```
displayMessage(['This section uses \slecompass \rmfunction to determine orientation of the sensor
              '\rmsystem object. Move the sensor to visualize orientation of the sensor in the figure window
              'Estimate Orientation using Ecompass algorithm.'])
tic;
viewer = HelperOrientationViewer('Title',{ 'Ecompass Algorithm'});
stopTimer = 100;
tic;
% Use ecompass algorithm to estimate orientation and update the viewer as the
% sensor moves for time specified by stopTimer.
while(toc < stopTimer)
    [accel,gyro,mag] = readSensorDataMPU9250(imu);
```

```
rotators = ecompass(accel,mag);  
for j = numel(rotators)  
    viewer(rotators(j));  
end  
end
```

Clean Up

When the connection is no longer needed, release and clear the objects

```
release(imu);  
delete(imu);  
clear;
```

Things to try

You can try this example with other sensors such as InvenSense MPU-6050 and STMicroelectronics LSM9DS1. Note that the MPU-6050 sensor can be used only with the `imufilter` system object.

Wireless Data Streaming and Sensor Fusion Using BNO055

This example shows how to get data from a Bosch BNO055 IMU sensor through an HC-05 Bluetooth® module, and to use the 9-axis AHRS fusion algorithm on the sensor data to compute orientation of the device. The example creates a figure which gets updated as you move the device.

BNO055 is a 9-axis sensor with accelerometer, gyroscope, and magnetometer. Accelerometer measures acceleration, gyroscope measures angular velocity, and magnetometer measures magnetic field in x -, y - and z - axes.

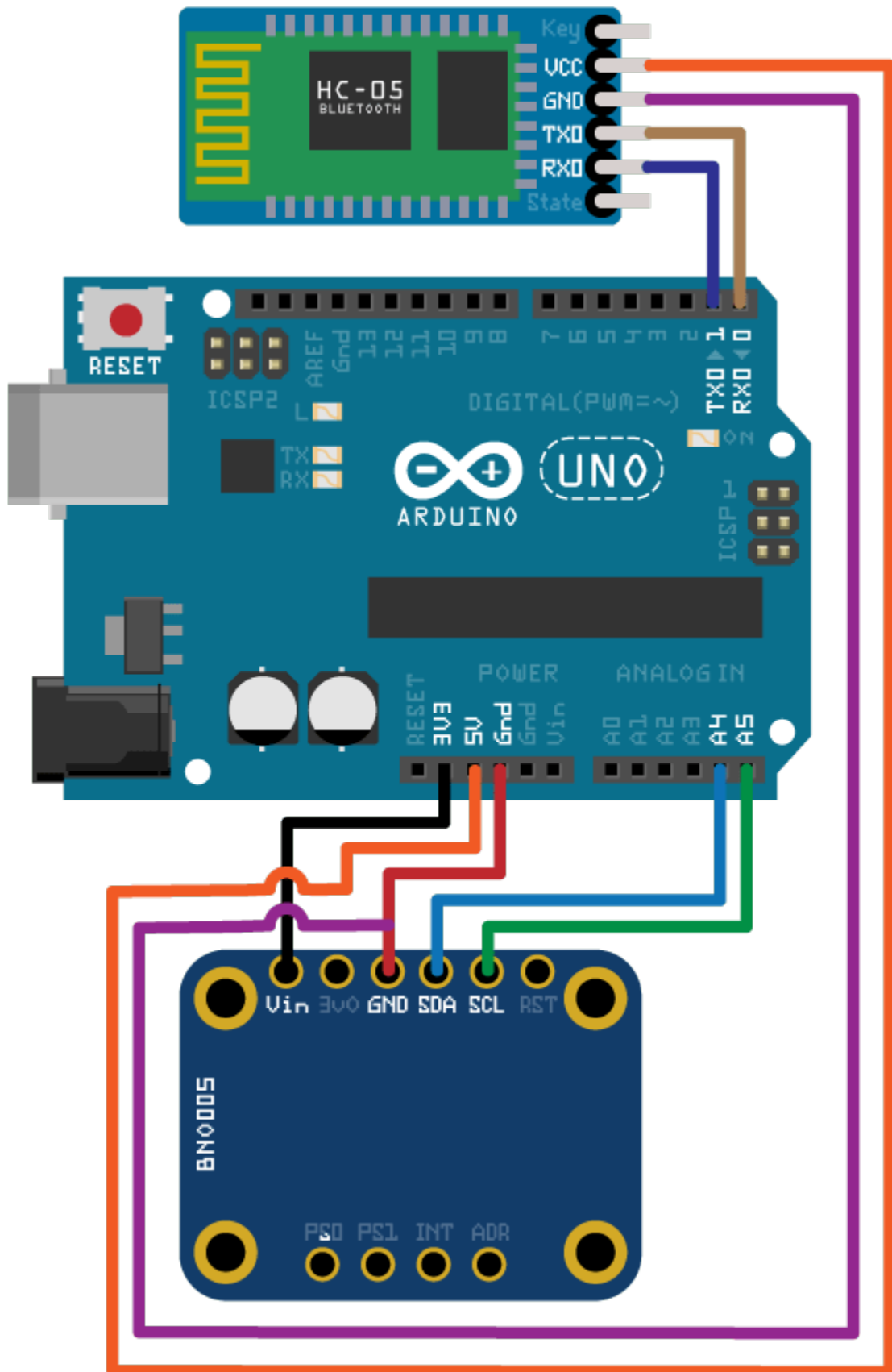
Required MathWorks® Products

- MATLAB®
- MATLAB Support Package for Arduino® Hardware
- Instrument Control Toolbox™
- Sensor Fusion and Tracking Toolbox™ or Navigation Toolbox™

Required Hardware

- Arduino® Uno
- Bosch BNO055 Sensor
- HC-05 Bluetooth® Module

Hardware Connection



Connect the SDA, SCL, GND, and the VCC pins of the BNO055 sensor to the corresponding pins on the Arduino® Uno board with the connections:

- SDA - A4
- SCL - A5
- VCC - +3.3V
- GND - GND

Connect the TX, RX, GND and VCC pins of the HC-05 module to the corresponding pins on the Arduino® Uno board. This example uses the connections:

- TX - Digital Pin 0 (RX)
- RX - Digital Pin 1 (TX)
- VCC - 5V
- GND - GND

Ensure that the connections to the sensors and Bluetooth® module are intact. It is recommended to use a BNO055 shield for Arduino Uno (Arduino 9 Axis Motion Shield). See “Troubleshooting Sensors” (MATLAB Support Package for Arduino Hardware) to debug the sensor related issues.

Setup and Configure Arduino for Bluetooth® Communication

Configure the Arduino Uno board to communicate through Bluetooth® using the `arduinsetup` command from MATLAB command prompt. See Setup and Configure Arduino Hardware for steps on how to configure the Arduino board for communication through Bluetooth®. Make sure to check the box for I2C Libraries to be included during the Setup.

Create Sensor Object

Create an `arduino` object.

```
a = arduino('btspp://98D33230EB9F', 'Uno');
```

Create the `bno055` sensor object in the `OperatingMode` 'amg'.

```
fs = 100; % Sample Rate in Hz
imu = bno055(a, 'SampleRate', fs, 'OutputFormat', 'matrix', 'OperatingMode', 'amg');
```

Compensating for Hard Iron and Soft Iron Distortions

Fusion algorithms use magnetometer readings which need to be compensated for magnetic distortions such as hard iron distortion. Hard iron distortions are produced by materials which create a magnetic field, resulting in shifting the origin on the response surface. These distortions can be corrected by subtracting the correction values from the magnetometer readings for each axis. In order to find the correction values,

- 1 Rotate the sensor from 0 to 360 degree along each axis.
- 2 Use the `magcal` function to obtain the correction coefficients.

These correction values change with the surroundings.

To obtain correction coefficients for both hard iron and soft iron distortions:

```
ts = tic;
stopTimer = 50;
```

```

magReadings=[];
while(toc(ts) < stopTimer)
    % Rotate the sensor along x axis from 0 to 360 degree.
    % Take 2-3 rotations to improve accuracy.
    % For other axes, rotate along that axes.
    [accel,gyro,mag] = read(imu);
    magReadings = [magReadings;mag];
end

[A, b] = magcal(magReadings); % A = 3x3 matrix for soft iron correction
                             % b = 3x1 vector for hard iron correction

```

Aligning the axis of BNO055 sensor with NED Coordinates

Sensor Fusion algorithms used in this example use North-East-Down (NED) as a fixed, parent coordinate system. In the NED reference frame, the x-axis points north, the y-axis points east, and the z-axis points down. Depending on the algorithm, north may either be the magnetic north or true north. The algorithms in this example use the magnetic north. The algorithms used here expect all the sensors in the object to have their axes aligned with NED convention. The sensor values have to be inverted so that they are in accordance with the NED coordinates.

Tuning Filter Parameters

The algorithms used in this example, when properly tuned, enable estimation of orientation and are robust against environmental noise sources. You must consider the situations in which the sensors are used and tune the filters accordingly. See “Custom Tuning of Fusion Filters” (Sensor Fusion and Tracking Toolbox) for more details related to tuning filter parameters.

The example uses `ahrsfilter` to demonstrate orientation estimation. See “Determine Orientation Using Inertial Sensors” (Sensor Fusion and Tracking Toolbox) for more details related to inertial fusion algorithms.

Accelerometer-Gyroscope-Magnetometer Fusion

An attitude and heading reference system (AHRS) consists of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute the orientation of the device. The `ahrsfilter` produces a smoothly changing estimate of orientation of the device, while correctly estimating the north direction. The `ahrsfilter` has the ability to remove gyroscope bias and can also detect and reject mild magnetic jamming.

The following code snippets use `ahrsfilter` system object to determine the orientation of the sensor and create a figure that gets updated as you move the sensor. The initial position of the sensor should be such that the device x-axis is pointing towards magnetic north, the device y-axis is pointing to east and the device z-axis is pointing downwards. You could use a cellphone or compass to determine magnetic north.

```

% GyroscopeNoise, AccelerometerNoise and MagnetometerNoise are determined from the BNO055 datasheet
% NoisePower = OutputNoisePowerDensityrms^2 * Bandwidth

GyroscopeNoiseBNO055 = 3.05e-06; % GyroscopeNoise (variance value) in units of (rad/s)^2
AccelerometerNoiseBNO055 = 67.53e-06; % AccelerometerNoise (variance value) in units of (m/s^2)^2
MagnetometerNoiseBNO055 = 1; %MagnetometerNoise (variance value) in units of uT^2

viewer = HelperOrientationViewer('Title',{'AHRS Filter'});

FUSE = ahrsfilter('SampleRate',imu.SampleRate,'GyroscopeNoise',GyroscopeNoiseBNO055,'AccelerometerNoise',AccelerometerNoiseBNO055,'MagnetometerNoise',MagnetometerNoiseBNO055);

```

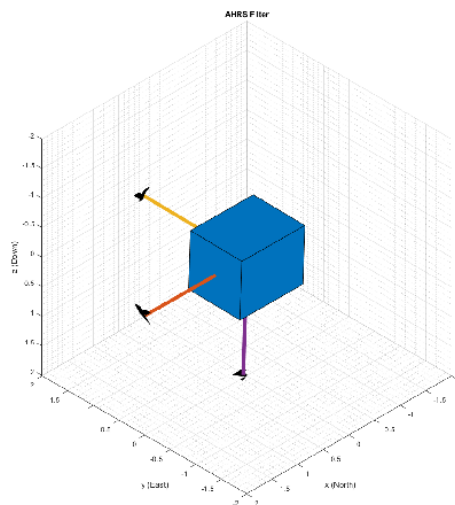
```
stopTimer=10;
```

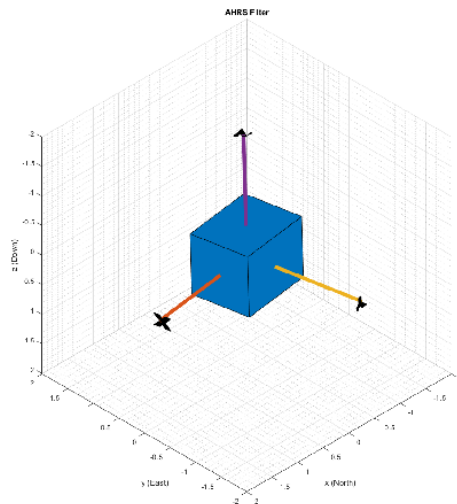
After executing the below code snippet, slowly move the sensor and check if the motion in the figure matches the motion of the sensor. Increase the `stopTimer` value, if you need to track the orientation for longer time.

```
magx_correction = b(1);
magy_correction = b(2);
magz_correction = b(3);
```

```
ts = tic;
while(toc(ts) < stopTimer)
    [accel,gyro,mag] = read(imu);
    % Align coordinates in accordance with NED convention
    accel = [-accel(:,1), accel(:,2), accel(:,3)];
    gyro = [gyro(:,1), -gyro(:,2), -gyro(:,3)];
    mag = [(mag(:,1)-magx_correction), -(mag(:,2)-magy_correction), -(mag(:,3)-magz_correction)];
    rotators = FUSE(accel,gyro,mag);
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

If the sensor is stationary at the initial position where the device x-axis points to the magnetic north, the device y-axis points to the east, and the device z-axis points downwards, the x-axis in the figure will be parallel to and aligned with the positive x-axis, the y-axis in the figure will be parallel to and aligned with the positive y-axis, and the z-axis in the figure will be parallel to and aligned with the positive z-axis.





Clean Up

When the connection is no longer needed, release and clear the objects.

```
release(imu);  
delete(imu);  
clear;
```

Things to try

You can try this example with other sensors such as InvenSense MPU-6050, MPU-9250, and STMicroelectronics LSM9DS1.

Rotations, Orientation, and Quaternions

This example reviews concepts in three-dimensional rotations and how quaternions are used to describe orientation and rotations. Quaternions are a skew field of hypercomplex numbers. They have found applications in aerospace, computer graphics, and virtual reality. In MATLAB®, quaternion mathematics can be represented by manipulating the `quaternion` class.

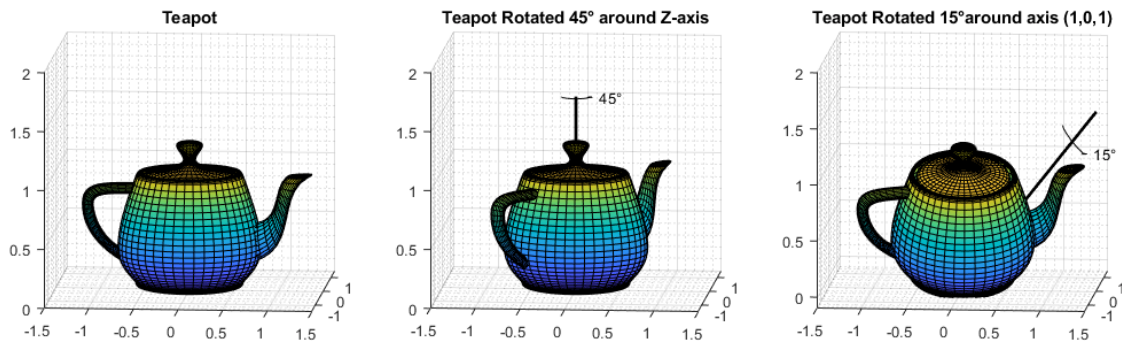
The `HelperDrawRotation` class is used to illustrate several portions of this example.

```
dr = HelperDrawRotation;
```

Rotations in Three Dimensions

All rotations in 3-D can be defined by an axis of rotation and an angle of rotation about that axis. Consider the 3-D image of a teapot in the leftmost plot. The teapot is rotated by 45 degrees around the Z-axis in the second plot. A more complex rotation of 15 degrees around the axis $[1\ 0\ 1]$ is shown in the third plot. Quaternions encapsulate the axis and angle of rotation and have an algebra for manipulating these rotations. The `quaternion` class, and this example, use the "right-hand rule" convention to define rotations. That is, positive rotations are clockwise around the axis of rotation when viewed from the origin.

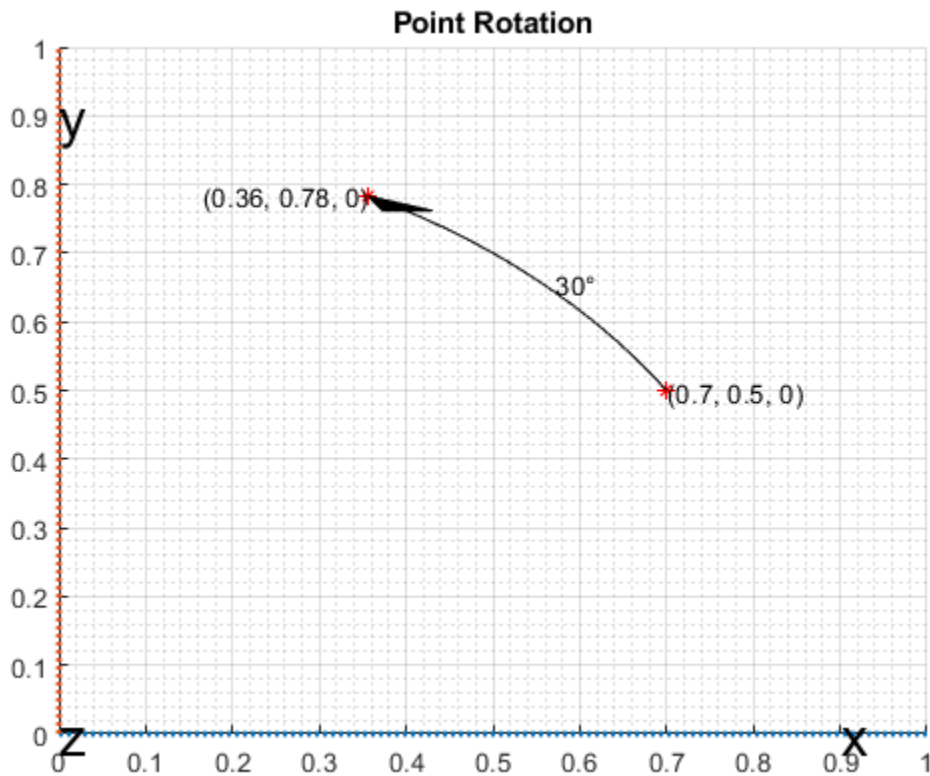
```
dr.drawTeapotRotations;
```



Point Rotation

The vertices of the teapot were rotated about the axis of rotation in the reference frame. Consider a point $(0.7, 0.5)$ rotated 30 degrees about the Z-axis.

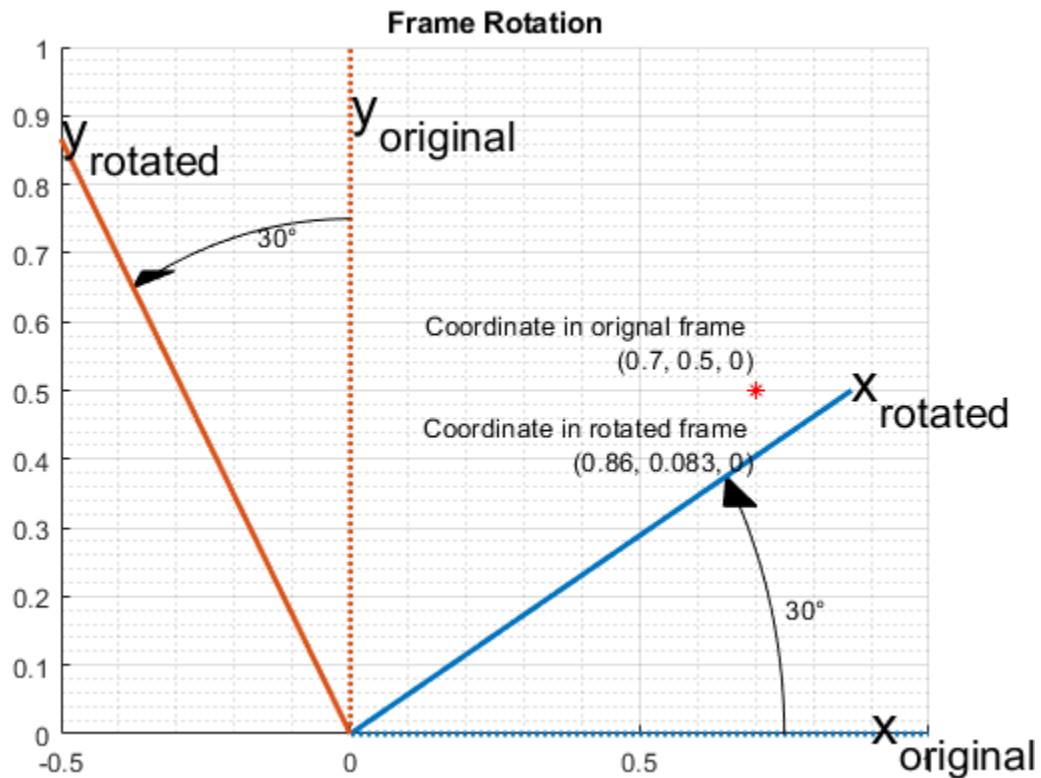
```
figure;
dr.draw2DPointRotation(gca);
```



Frame Rotation

Frame rotation is, in some sense, the opposite of point rotation. In frame rotation, the points of the object stay fixed, but the frame of reference is rotated. Again, consider the point (0.7, 0.5). Now the reference frame is rotated by 30 degrees around the Z-axis. Note that while the point (0.7, 0.5) stays fixed, it has different coordinates in the new, rotated frame of reference.

```
figure;  
dr.draw2DFrameRotation(gca);
```

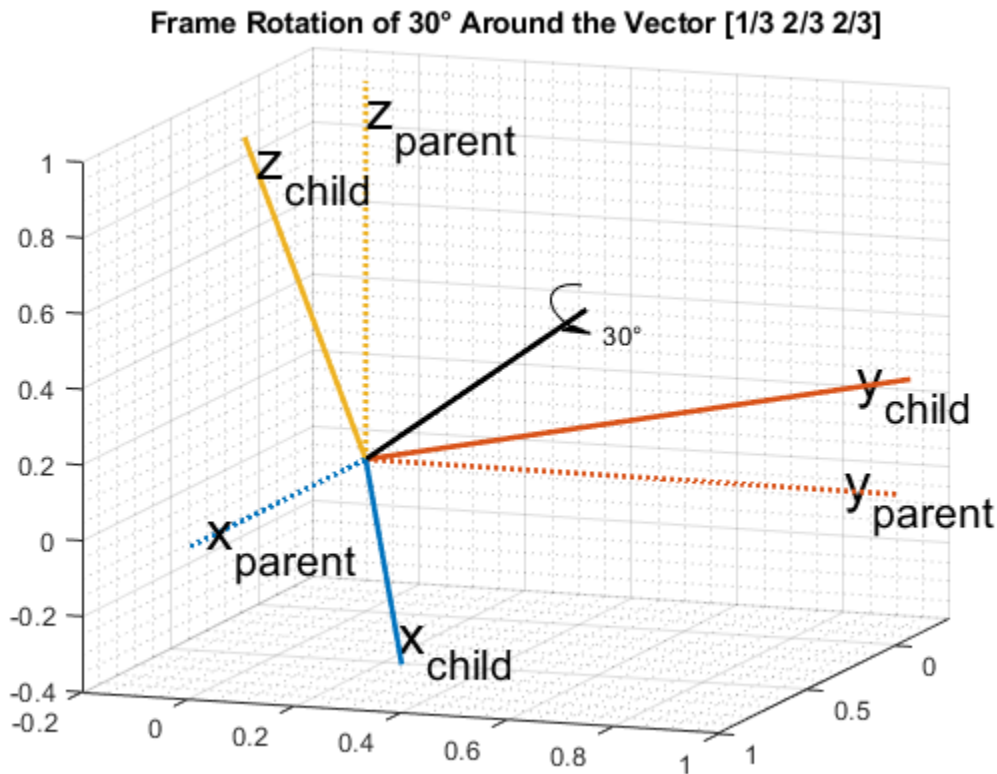



Orientation

Orientation refers to the angular displacement of an object relative to a frame of reference. Typically, orientation is described by the rotation that causes this angular displacement from a starting orientation. In this example, orientation is defined as the rotation that takes a quantity in a parent reference frame to a child reference frame. Orientation is usually given as a quaternion, rotation matrix, set of Euler angles, or rotation vector. It is useful to think about orientation as a frame rotation: the child reference frame is rotated relative to the parent frame.

Consider an example where the child reference frame is rotated 30 degrees around the vector $[1/3 \ 2/3 \ 2/3]$.

```
figure;
dr.draw3DOrientation(gca, [1/3 2/3 2/3], 30);
```



Quaternions

Quaternions are numbers of the form

$$a + bi + cj + dk$$

where

$$i^2 = j^2 = k^2 = ijk = -1$$

and $a, b, c,$ and d are real numbers. In the rest of this example, the four numbers $a, b, c,$ and d are referred to as the *parts* of the quaternion.

Quaternions for Rotations and Orientation

The axis and the angle of rotation are encapsulated in the quaternion parts. For a unit vector axis of rotation $[x, y, z]$, and rotation angle α , the quaternion describing this rotation is

$$\cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(xi + yj + zk)$$

Note that to describe a rotation using a quaternion, the quaternion must be a *unit quaternion*. A unit quaternion has a norm of 1, where the norm is defined as

$$\text{norm}(q) = \sqrt{a^2 + b^2 + c^2 + d^2}$$

There are a variety of ways to construct a quaternion in MATLAB, for example:

```
q1 = quaternion(1,2,3,4)
```

```
q1 =
```

```
    quaternion
```

```
    1 + 2i + 3j + 4k
```

Arrays of quaternions can be made in the same way:

```
quaternion([1 10; -1 1], [2 20; -2 2], [3 30; -3 3], [4 40; -4 4])
```

```
ans =
```

```
    2x2 quaternion array
```

```
    1 + 2i + 3j + 4k    10 + 20i + 30j + 40k
   -1 - 2i - 3j - 4k    1 + 2i + 3j + 4k
```

Arrays with four columns can also be used to construct quaternions, with each column representing a quaternion part:

```
qmgk = quaternion(magic(4))
```

```
qmgk =
```

```
    4x1 quaternion array
```

```
    16 + 2i + 3j + 13k
     5 + 11i + 10j + 8k
     9 + 7i + 6j + 12k
     4 + 14i + 15j + 1k
```

Quaternions can be indexed and manipulated just like any other array:

```
qmgk(3)
```

```
ans =
```

```
    quaternion
```

```
    9 + 7i + 6j + 12k
```

```
reshape(qmgk,2,2)
```

```
ans =
```

```
2x2 quaternion array
```

```
16 + 2i + 3j + 13k    9 + 7i + 6j + 12k  
5 + 11i + 10j + 8k    4 + 14i + 15j + 1k
```

```
[q1; q1]
```

```
ans =
```

```
2x1 quaternion array
```

```
1 + 2i + 3j + 4k  
1 + 2i + 3j + 4k
```

Quaternion Math

Quaternions have well-defined arithmetic operations. Addition and subtraction are similar to complex numbers: parts are added/subtracted independently. Multiplication is more complicated because of the earlier equation:

$$i^2 = j^2 = k^2 = ijk = -1$$

This means that multiplication of quaternions is not commutative. That is, $pq \neq qp$ for quaternions P and Q . However, every quaternion has a multiplicative inverse, so quaternions can be divided. Arrays of the quaternion class can be added, subtracted, multiplied, and divided in MATLAB.

```
q = quaternion(1,2,3,4);  
p = quaternion(-5,6,-7,8);
```

Addition

```
p + q
```

```
ans =
```

```
quaternion
```

```
-4 + 8i - 4j + 12k
```

Subtraction

```
p - q
```

```
ans =
```

```
quaternion
```

$$-6 + 4i - 10j + 4k$$

Multiplication

$p * q$

ans =

quaternion

$$-28 - 56i - 30j + 20k$$

Multiplication in the reverse order (note the different result)

$q * p$

ans =

quaternion

$$-28 + 48i - 14j - 44k$$

Right division of p by q is equivalent to $p(q^{-1})$.

$p ./ q$

ans =

quaternion

$$0.6 + 2.2667i + 0.53333j - 0.13333k$$

Left division of q by p is equivalent to $p^{-1}q$.

$p \setminus q$

ans =

quaternion

$$0.10345 + 0.2069i + 0j - 0.34483k$$

The conjugate of a quaternion is formed by negating each of the non-real parts, similar to conjugation for a complex number:

```
conj(p)
```

```
ans =
```

```
quaternion
```

```
-5 - 6i + 7j - 8k
```

Quaternions can be normalized in MATLAB:

```
pnormed = normalize(p)
```

```
pnormed =
```

```
quaternion
```

```
-0.37905 + 0.45486i - 0.53067j + 0.60648k
```

```
norm(pnormed)
```

```
ans =
```

```
1
```

Point and Frame Rotations with Quaternions

Quaternions can be used to rotate points in a static frame of reference, or to rotate the frame of reference itself. The `rotatepoint` function rotates a point $v = (v_x, v_y, v_z)$ using a quaternion q through the following equation:

$$pv_{quat}p^*$$

where v_{quat} is

$$v_{quat} = 0 + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$$

and P^* indicates quaternion conjugation. Note the above quaternion multiplication results in a quaternion with the real part, a , equal to 0. The b , c , and d parts of the result form the rotated point (b, c, d) .

Consider the example of point rotation from above. The point (0.7, 0.5) was rotated 30 degrees around the Z-axis. In three dimensions this point has a 0 Z-coordinate. Using the axis-angle formulation, a quaternion can be constructed using [0 0 1] as the axis of rotation.

```
ang = deg2rad(30);  
q = quaternion(cos(ang/2), 0, 0, sin(ang/2));  
pt = [0.7, 0.5, 0]; % Z-coordinate is 0 in the X-Y plane  
ptrot = rotatepoint(q, pt)
```

```
ptrot =
    0.3562    0.7830    0
```

Similarly, the `rotateframe` function takes a quaternion q and point v to compute

$$p^* v_{quat} p$$

Again the above quaternion multiplication results in a quaternion with 0 real part. The (b, c, d) parts of the result form the coordinate of the point v in the new, rotated reference frame. Using the quaternion class:

```
ptframerot = rotateframe(q, pt)
```

```
ptframerot =
    0.8562    0.0830    0
```

A quaternion and its conjugate have opposite effects because of the symmetry in the point and frame rotation equations. Rotating by the conjugate "undoes" the rotation.

```
rotateframe(conj(q), ptframerot)
```

```
ans =
    0.7000    0.5000    0
```

Because of the symmetry of the equations, this code performs the same rotation.

```
rotatepoint(q, ptframerot)
```

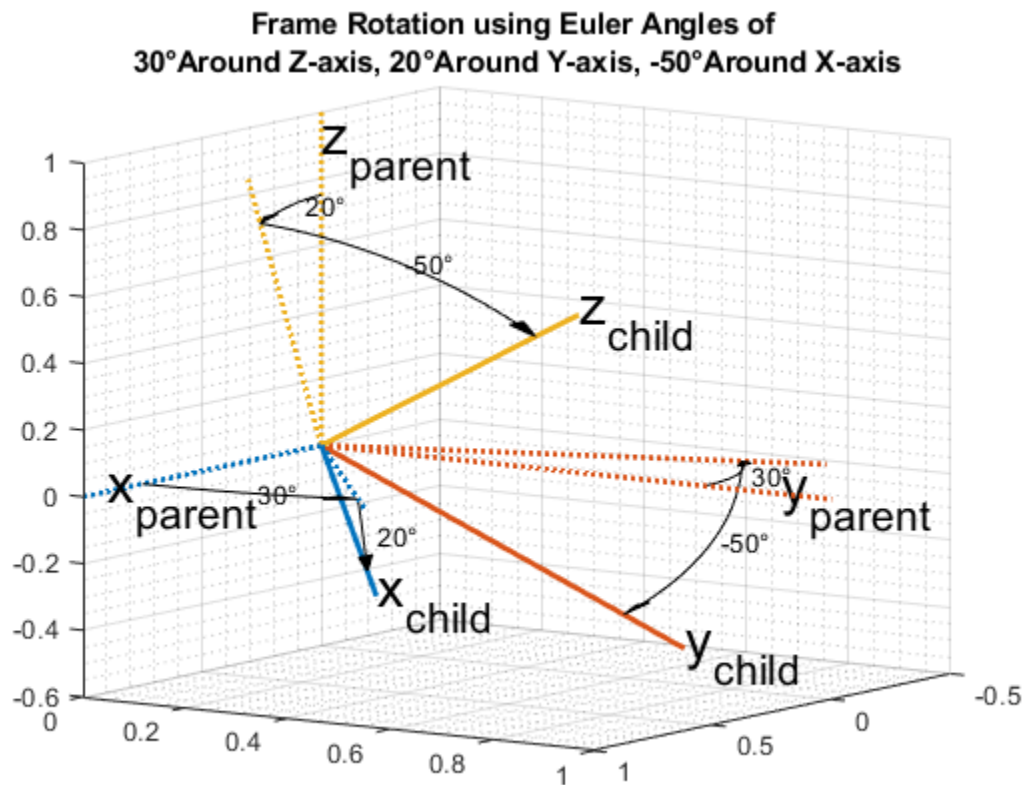
```
ans =
    0.7000    0.5000    0
```

Other Rotation Representations

Often rotations and orientations are described using alternate means: Euler angles, rotation matrices, and/or rotation vectors. All of these interoperate with quaternions in MATLAB.

Euler angles are frequently used because they are easy to interpret. Consider a frame of reference rotated by 30 degrees around the Z-axis, then 20 degrees around the Y-axis, and then -50 degrees around the X-axis. Note here, and throughout, the rotations around each axis are *intrinsic*: each subsequent rotation is around the newly created set of axes. In other words, the second rotation is around the "new" Y-axis created by the first rotation, not around the original Y-axis.

```
figure;
euld = [30 20 -50];
dr.drawEulerRotation(gca, euld);
```



To build a quaternion from these Euler angles for the purpose of frame rotation, use the `quaternion` constructor. Since the order of rotations is around the Z-axis first, then around the new Y-axis, and finally around the new X-axis, use the 'ZYX' flag.

```
qeul = quaternion(deg2rad(euld), 'euler', 'ZYX', 'frame')
```

```
qeul =
```

```
quaternion
```

```
0.84313 - 0.44275i + 0.044296j + 0.30189k
```

The 'euler' flag indicates that the first argument is in radians. If the argument is in degrees, use the 'eulerd' flag.

```
qeuld = quaternion(euld, 'eulerd', 'ZYX', 'frame')
```

```
qeuld =
```

```
quaternion
```

```
0.84313 - 0.44275i + 0.044296j + 0.30189k
```


To convert back to Euler angles:

```
rad2deg(euler(qeul, 'ZYX', 'frame'))
```

```
ans =
```

```
30.0000 20.0000 -50.0000
```

Equivalently, the `eulerd` method can be used.

```
eulerd(qeul, 'ZYX', 'frame')
```

```
ans =
```

```
30.0000 20.0000 -50.0000
```

Alternatively, this same rotation can be represented as a rotation matrix:

```
rmat = rotmat(qeul, 'frame')
```

```
rmat =
```

```
0.8138 0.4698 -0.3420
-0.5483 0.4257 -0.7198
-0.1926 0.7733 0.6040
```

The conversion back to quaternions is similar:

```
quaternion(rmat, 'rotmat', 'frame')
```

```
ans =
```

```
quaternion
```

```
0.84313 - 0.44275i + 0.044296j + 0.30189k
```

Just as a quaternion can be used for either point or frame rotation, it can be converted to a rotation matrix (or set of Euler angles) specifically for point or frame rotation. The rotation matrix for point rotation is the transpose of the matrix for frame rotation. To convert between rotation representations, it is necessary to specify 'point' or 'frame'.

The rotation matrix for the point rotation section of this example is:

```
rotmatPoint = rotmat(q, 'point')
```

```
rotmatPoint =
```

```
0.8660 -0.5000 0
0.5000 0.8660 0
```

```
0 0 1.0000
```

To find the location of the rotated point, right-multiply `rotmatPoint` by the transposed array `pt`.

```
rotmatPoint * (pt')
```

```
ans =
```

```
0.3562  
0.7830  
0
```

The rotation matrix for the frame rotation section of this example is:

```
rotmatFrame = rotmat(q, 'frame')
```

```
rotmatFrame =
```

```
0.8660 0.5000 0  
-0.5000 0.8660 0  
0 0 1.0000
```

To find the location of the point in the rotated reference frame, right-multiply `rotmatFrame` by the transposed array `pt`.

```
rotmatFrame * (pt')
```

```
ans =
```

```
0.8562  
0.0830  
0
```

A rotation vector is an alternate, compact rotation encapsulation. A rotation vector is simply a three-element vector that represents the unit length axis of rotation scaled-up by the angle of rotation in radians. There is no frame-ness or point-ness associated with a rotation vector. To convert to a rotation vector:

```
rv = rotvec(qeul)
```

```
rv =
```

```
-0.9349 0.0935 0.6375
```

To convert to a quaternion:

```
quaternion(rv, 'rotvec')
```

```
ans =
```

```
quaternion
```

```
0.84313 - 0.44275i + 0.044296j + 0.30189k
```

Distance

One advantage of quaternions over Euler angles is the lack of discontinuities. Euler angles have discontinuities that vary depending on the convention being used. The `dist` function compares the effect of rotation by two different quaternions. The result is a number in the range of 0 to π .

Consider two quaternions constructed from Euler angles:

```
eul1 = [0, 10, 0];
eul2 = [0, 15, 0];
qdist1 = quaternion(deg2rad(eul1), 'euler', 'ZYX', 'frame');
qdist2 = quaternion(deg2rad(eul2), 'euler', 'ZYX', 'frame');
```

Subtracting the Euler angles, you can see there is no rotation around the Z-axis or X-axis.

```
eul2 - eul1
```

```
ans =
```

```
0 5 0
```

The difference between these two rotations is five degrees around the Y-axis. The `dist` shows the difference as well.

```
rad2deg(dist(qdist1, qdist2))
```

```
ans =
```

```
5.0000
```

For Euler angles such as `eul1` and `eul2`, computing angular distance is trivial. A more complex example, which spans an Euler angle discontinuity, is:

```
eul3 = [0, 89, 0];
eul4 = [180, 89, 180];
qdist3 = quaternion(deg2rad(eul3), 'euler', 'ZYX', 'frame');
qdist4 = quaternion(deg2rad(eul4), 'euler', 'ZYX', 'frame');
```

Though `eul3` and `eul4` represent nearly the same orientation, simple Euler angle subtraction gives the impression that these two orientations are very far apart.

```
euldiff = eul4 - eul3
```

```
euldiff =
```

```
180 0 180
```

Using the `dist` function on the quaternions shows that there is only a two-degree difference in these rotations:

```
euldist = rad2deg(dist(qdist3, qdist4))
```

```
euldist =  
    2.0000
```

A quaternion and its negative represent the same rotation. This is not obvious from subtracting quaternions, but the `dist` function makes it clear.

```
qpos = quaternion(-cos(pi/4), 0, 0, sin(pi/4))
```

```
qpos =  
    quaternion  
    -0.70711 +      0i +      0j + 0.70711k
```

```
qneg = -qpos
```

```
qneg =  
    quaternion  
    0.70711 +      0i +      0j - 0.70711k
```

```
qdiff = qpos - qneg
```

```
qdiff =  
    quaternion  
    -1.4142 +      0i +      0j + 1.4142k
```

```
dist(qpos, qneg)
```

```
ans =  
    0
```

Supported Functions

The `quaternion` class lets you effectively describe rotations and orientations in MATLAB. The full list of quaternion-supported functions can be found with the `methods` function:

```
methods('quaternion')
```

Methods for class quaternion:

```
angvel          ismatrix      prod
cat             isnan         quaternion
classUnderlying isrow         rdivide
compact        isscalar      reshape
conj           isvector      rotateframe
ctranspose     ldivide       rotatepoint
disp           length        rotmat
dist          log          rotvec
double        meanrot      rotvecd
eq            minus        single
euler         mtimes       size
eulerd       ndims        slerp
exp          ne          times
horzcat      norm         transpose
iscolumn     normalize    uminus
isempty      numel        validateattributes
isequal      parts        vertcat
isequaln    permute
isfinite     plus
isinf       power
```

Static methods:

```
ones          zeros
```

Lowpass Filter Orientation Using Quaternion SLERP

This example shows how to use spherical linear interpolation (SLERP) to create sequences of quaternions and lowpass filter noisy trajectories. SLERP is a commonly used computer graphics technique for creating animations of a rotating object.

SLERP Overview

Consider a pair of quaternions q_0 and q_1 . Spherical linear interpolation allows you to create a sequence of quaternions that vary smoothly between q_0 and q_1 with a constant angular velocity. SLERP uses an interpolation parameter h that can vary between 0 and 1 and determines how close the output quaternion is to either q_0 or q_1 .

The original formulation of quaternion SLERP was given by Ken Shoemake [1] as:

$$\text{Slerp}(q_0, q_1, h) = q_1(q_1^{-1}q_0)^h$$

An alternate formulation with sinusoids (used in the `slerp` function implementation) is:

$$\text{Slerp}(q_0, q_1, h) = \frac{\sin((1-h)\theta)}{\sin\theta}q_0 + \frac{\sin(h\theta)}{\sin\theta}q_1$$

where θ is the dot product of the quaternion parts. Note that $\theta = \text{dist}(q_0, q_1)/2$.

SLERP vs Linear Interpolation of Quaternion Parts

Consider the following example. Build two quaternions from Euler angles.

```
q0 = quaternion([-80 10 0], 'eulerd', 'ZYX', 'frame');
q1 = quaternion([80 70 70], 'eulerd', 'ZYX', 'frame');
```

To find a quaternion 30 percent of the way from q_0 to q_1 , specify the `slerp` parameter as 0.3.

```
p30 = slerp(q0, q1, 0.3);
```

To view the interpolated quaternion's Euler angle representation, use the `eulerd` function.

```
eulerd(p30, 'ZYX', 'frame')
```

```
ans =
```

```
-56.6792    33.2464   -9.6740
```

To create a smooth trajectory between q_0 and q_1 , specify the `slerp` interpolation parameter as a vector of evenly spaced numbers between 0 and 1.

```
dt = 0.01;
h = (0:dt:1).';
trajSlerped = slerp(q0, q1, h);
```

Compare the results of the SLERP algorithm with a trajectory between q_0 and q_1 , using simple linear interpolation (LERP) of each quaternion part.

```
partsLinInterp = interp1([0;1], compact([q0;q1]), h, 'linear');
```

Note that linear interpolation does not give unit quaternions, so they must be normalized.

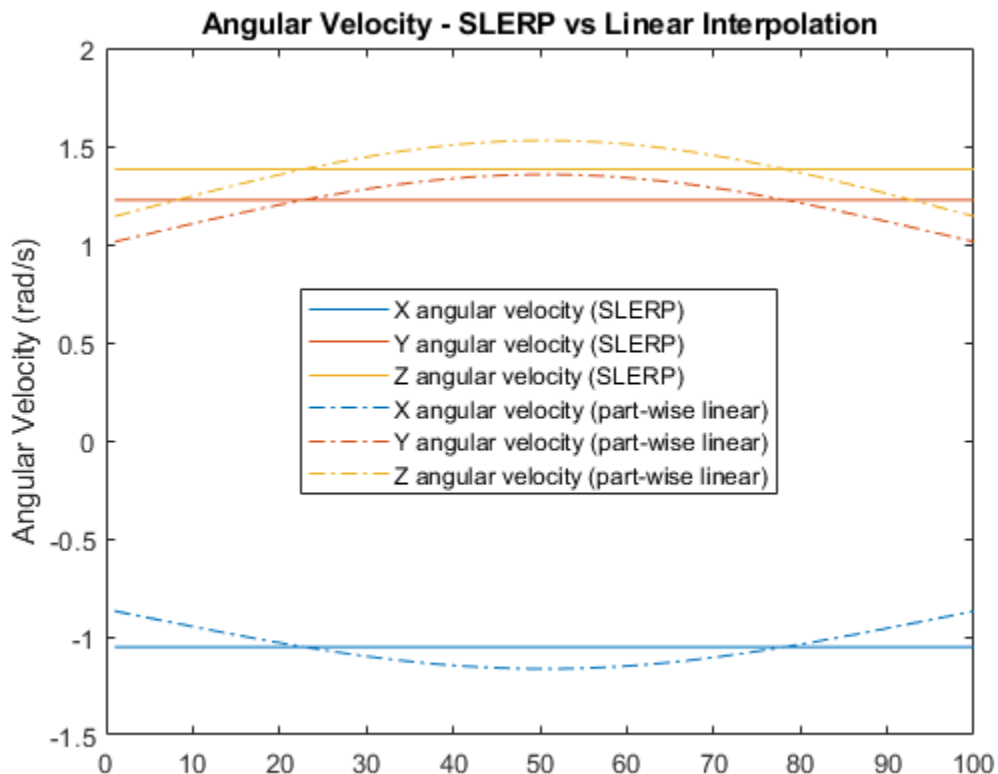
```
trajLerped = normalize(quaternion(partsLinInterp));
```

Compute the angular velocities from each approach.

```
avSlerp = helperQuat2AV(trajSlerped, dt);
avLerp = helperQuat2AV(trajLerped, dt);
```

Plot both sets of angular velocities. Notice that the angular velocity for SLERP is constant, but it varies for linear interpolation.

```
sp = HelperSlerpPlotting;
sp.plotAngularVelocities(avSlerp, avLerp);
```



SLERP produces a smooth rotation at a constant rate.

Lowpass Filtering with SLERP

SLERP can also be used to make more complex functions. Here, SLERP is used to lowpass filter a noisy trajectory.

Rotational noise can be constructed by forming a quaternion from a noisy rotation vector.

```
rcurr = rng(1);
sigma = 1e-1;
noiserv = sigma .* ( rand(numel(h), 3) - 0.5);
```

```
qnoise = quaternion(noiserv, 'rotvec');  
rng(rcurr);
```

To corrupt the trajectory `trajSlerped` with noise, incrementally rotate the trajectory with the noise vector `qnoise`.

```
trajNoisy = trajSlerped .* qnoise;
```

You can smooth real-valued signals using a single pole filter of the form:

$$y_k = y_{k-1} + \alpha(x_k - y_{k-1})$$

This formula essentially says that the new filter state y_k should be moved toward the current input x_k by a step size that is proportional to the distance between the current input and the current filter state y_{k-1} .

The spirit of this approach informs how a quaternion sequence can be lowpass filtered. To do this, both the `dist` and `slerp` functions are used.

The `dist` function returns a measurement in radians of the difference in rotation applied by two quaternions. The range of the `dist` function is the half-open interval $[0, \pi)$.

The `slerp` function is used to steer the filter state towards the current input. It is steered more towards the input when the difference between the input and current filter state has a large `dist`, and less toward the input when `dist` gives a small value. The interpolation parameter to `slerp` is in the closed-interval $[0,1]$, so the output of `dist` must be re-normalized to this range. However, the full range of $[0,1]$ for the interpolation parameter gives poor performance, so it is limited to a smaller range `hrange` centered at `hbias`.

```
hrange = 0.4;  
hbias = 0.4;
```

Limit `low` and `high` to the interval $[0, 1]$.

```
low = max(min(hbias - (hrange./2), 1), 0);  
high = max(min(hbias + (hrange./2), 1), 0);  
hrangeLimited = high - low;
```

Initialize the filter and preallocate outputs.

```
y = trajNoisy(1); % initial filter state  
qout = zeros(size(y), 'like', y); % preallocate filter output  
qout(1) = y;
```

Filter the noisy trajectory, sample-by-sample.

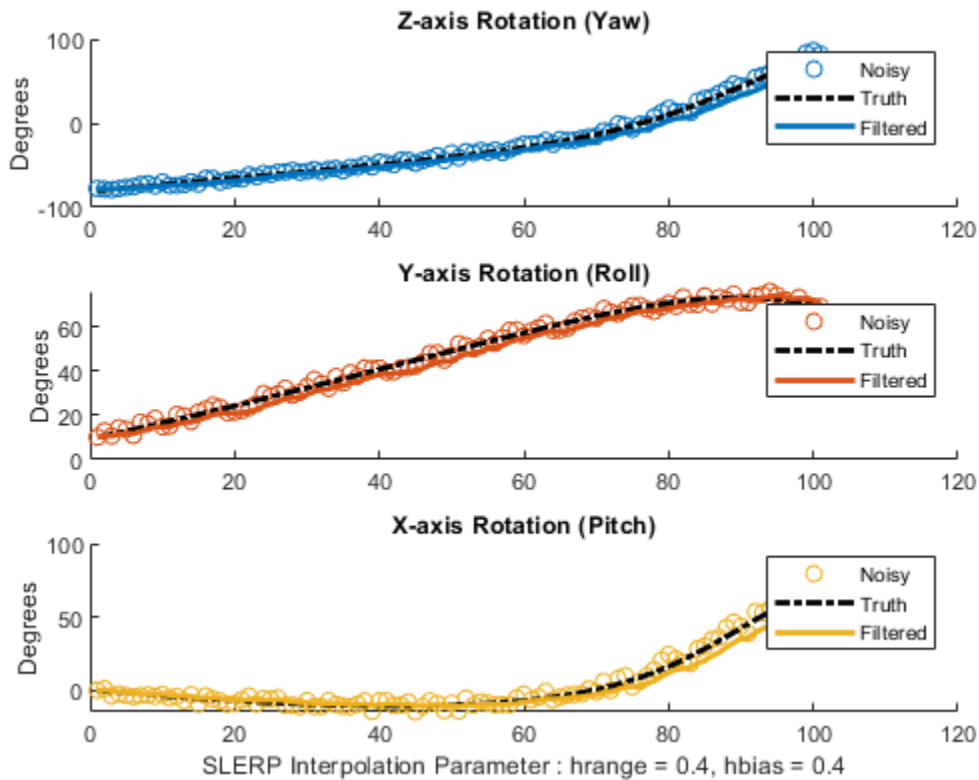
```
for ii=2:numel(trajNoisy)  
    x = trajNoisy(ii);  
    d = dist(y, x);  
  
    % Renormalize dist output to the range [low, high]  
    hlpf = (d./pi).*hrangeLimited + low;  
    y = slerp(y,x,hlpf);  
    qout(ii) = y;  
end  
f = figure;
```



```

sp.plotEulerd(f, trajNoisy, 'o');
sp.plotEulerd(f, trajSlerped, 'k-.', 'LineWidth', 2);
sp.plotEulerd(f, qout, '-', 'LineWidth', 2);
sp.addAnnotations(f, hrange, hbias);

```



Conclusion

SLERP can be used for creating both short trajectories between two orientations and for smoothing or lowpass filtering. It has found widespread use in a variety of industries.

References

- 1 Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* 19, no 3 (1985):245-54, doi:10.1145/325165.325242

Introduction to Simulating IMU Measurements

This example shows how to simulate inertial measurement unit (IMU) measurements using the `imuSensor` (Sensor Fusion and Tracking Toolbox) System object. An IMU can include a combination of individual sensors, including a gyroscope, an accelerometer, and a magnetometer. You can specify properties of the individual sensors using `gyroparams` (Sensor Fusion and Tracking Toolbox), `accelparams` (Sensor Fusion and Tracking Toolbox), and `magparams` (Sensor Fusion and Tracking Toolbox), respectively.

In the following plots, unless otherwise noted, only the x-axis measurements are shown. Use the sliders to interactively tune the parameters.

Default Parameters

The default parameters for the gyroscope model simulate an ideal signal. Given a sinusoidal input, the gyroscope output should match exactly.

```
params = gyroparams
```

```
params =
```

```
gyroparams with properties:
```

```

MeasurementRange: Inf          rad/s
Resolution: 0                (rad/s)/LSB
ConstantBias: [0 0 0]        rad/s
AxesMisalignment: [3x3 double] %
NoiseDensity: [0 0 0]        (rad/s)/√Hz
BiasInstability: [0 0 0]     rad/s
RandomWalk: [0 0 0]          (rad/s)*√Hz

TemperatureBias: [0 0 0]     (rad/s)/°C
TemperatureScaleFactor: [0 0 0] %/°C
AccelerationBias: [0 0 0]    (rad/s)/(m/s2)

```

```
% Generate N samples at a sampling rate of Fs with a sinusoidal frequency
% of Fc.
```

```
N = 1000;
```

```
Fs = 100;
```

```
Fc = 0.25;
```

```
t = (0:(1/Fs):((N-1)/Fs)).';
```

```
acc = zeros(N, 3);
```

```
angvel = zeros(N, 3);
```

```
angvel(:,1) = sin(2*pi*Fc*t);
```

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
```

```
[~, gyroData] = imu(acc, angvel);
```

```
figure
```

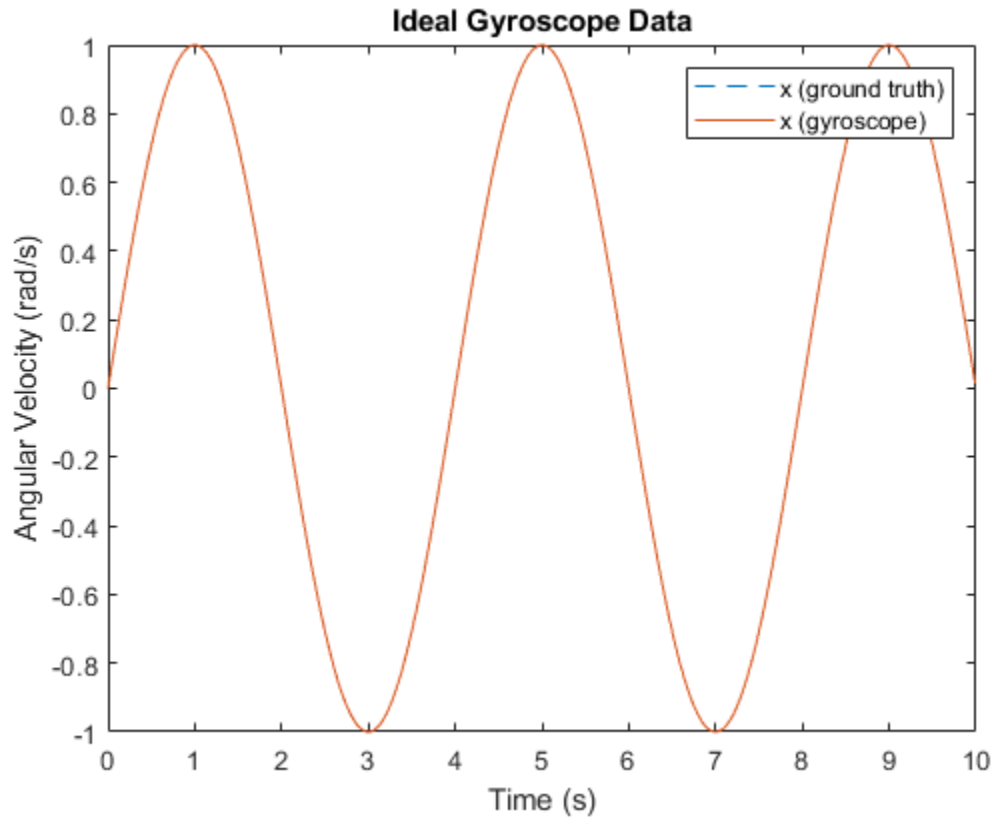
```
plot(t, angvel(:,1), '--', t, gyroData(:,1))
```

```
xlabel('Time (s)')
```

```
ylabel('Angular Velocity (rad/s)')
```

```
title('Ideal Gyroscope Data')
```

```
legend('x (ground truth)', 'x (gyroscope)')
```



Hardware Parameter Tuning

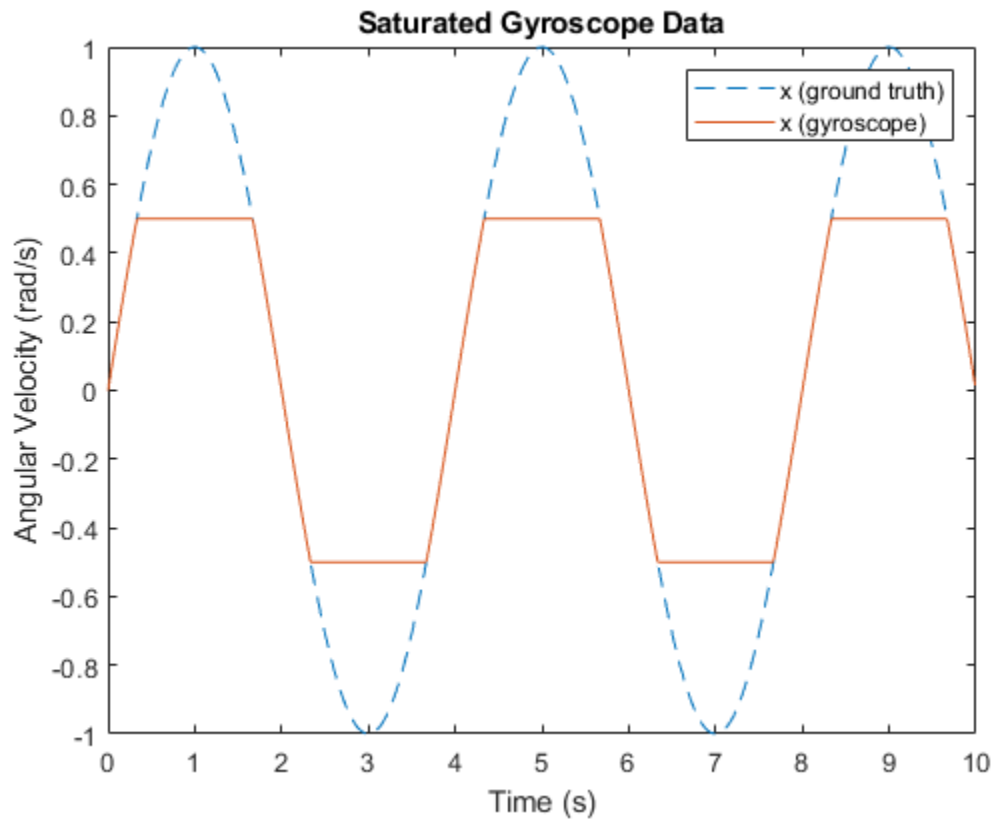
The following parameters model hardware limitations or defects. Some can be corrected through calibration.

`MeasurementRange` determines the maximum absolute value reported by the gyroscope. Larger absolute values are saturated. The effect is shown by setting the measurement range to a value smaller than the amplitude of the sinusoidal ground-truth angular velocity.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.MeasurementRange = 0.5  ; % rad/s

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Saturated Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

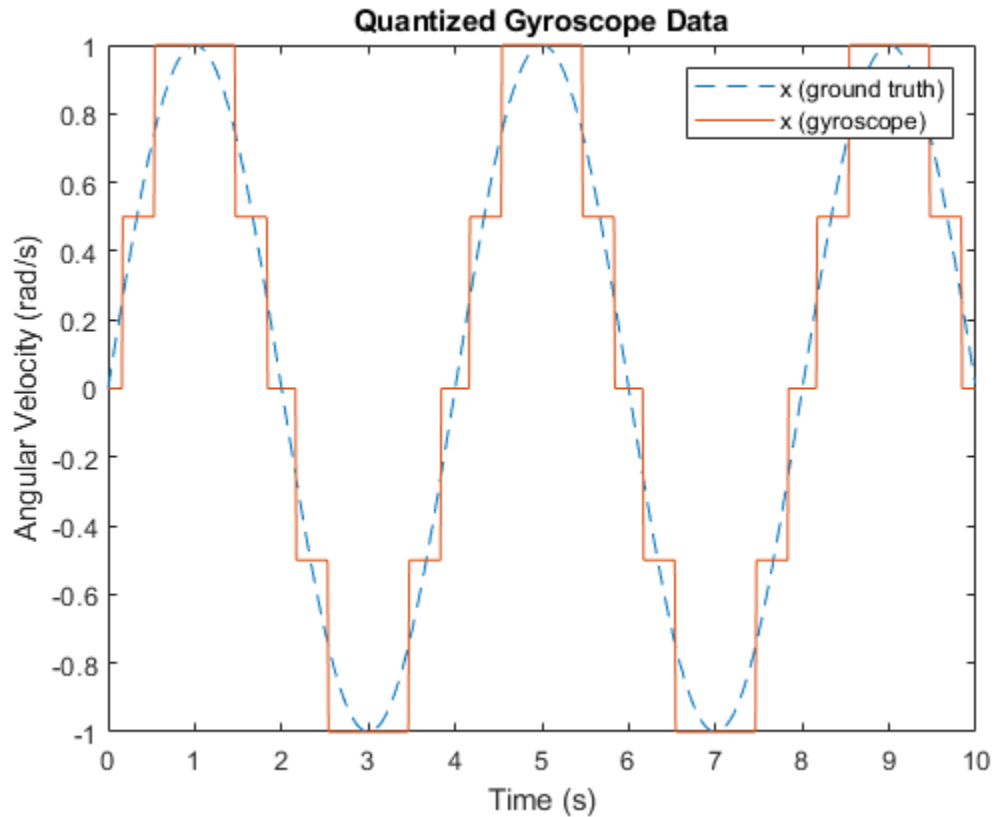


Resolution affects the step size of the digital measurements. Use this parameter to model the quantization effects from the analog-to-digital converter (ADC). The effect is shown by increasing the parameter to a much larger value than is typical.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.Resolution = 0.5  ; % (rad/s)/LSB

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Quantized Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

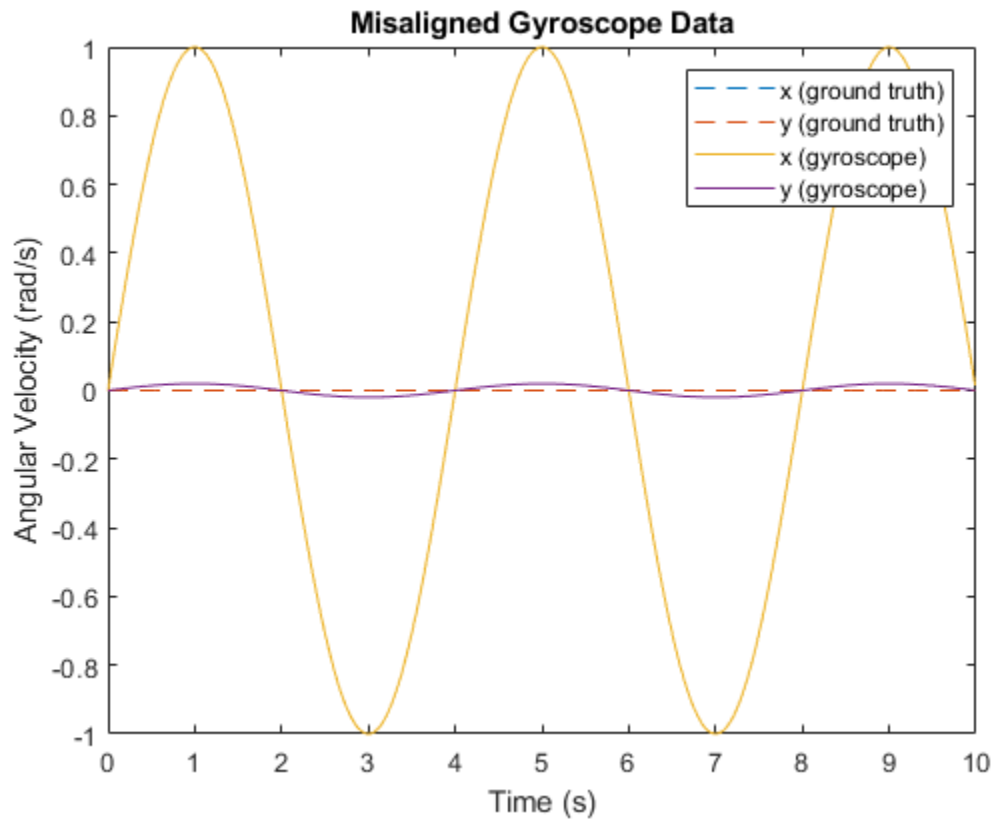


AxesMisalignment is the amount of skew in the sensor axes. This skew normally occurs when the sensor is mounted to the PCB and can be corrected through calibration. The effect is shown by skewing the x-axis slightly and plotting both the x-axis and y-axis.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
xMisalignment = 2 ; % percent
imu.Gyroscope.AxesMisalignment = [xMisalignment, 0, 0]; % percent

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1:2), '--', t, gyroData(:,1:2))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Misaligned Gyroscope Data')
legend('x (ground truth)', 'y (ground truth)', ...
       'x (gyroscope)', 'y (gyroscope)')
```

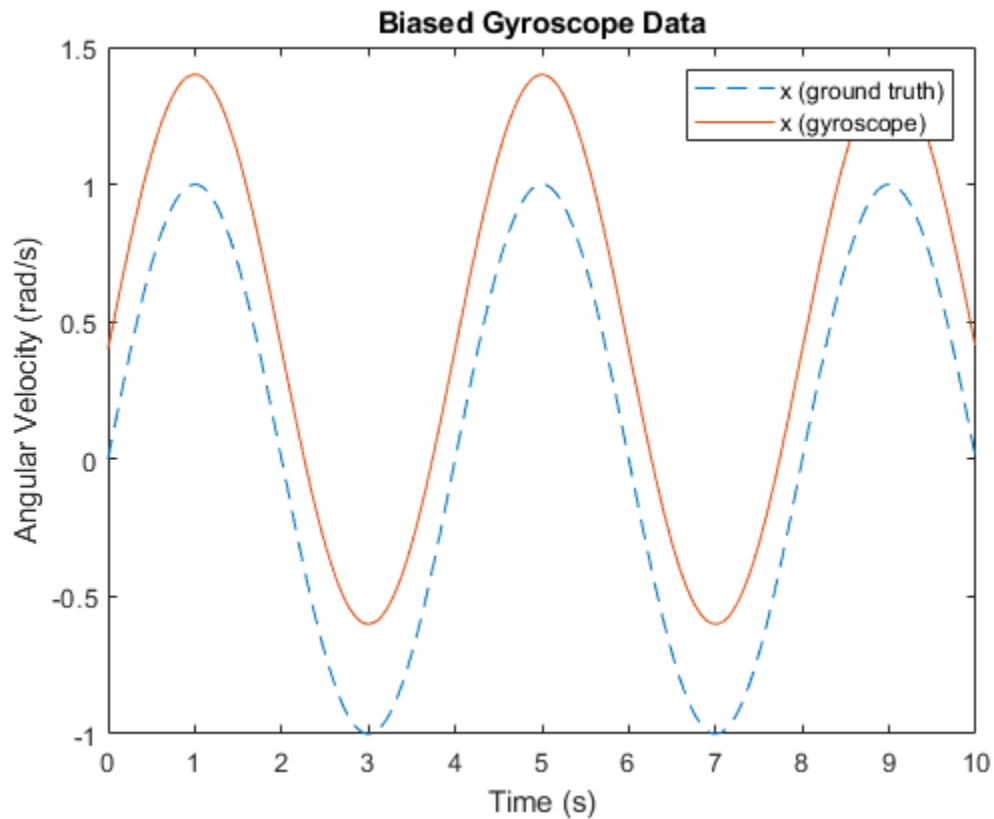


ConstantBias occurs in sensor measurements due to hardware defects. Since this bias is not caused by environmental factors, such as temperature, it can be corrected through calibration.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
xBias = 0.4 ; % rad/s
imu.Gyroscope.ConstantBias = [xBias, 0, 0]; % rad/s

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Biased Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```




Random Noise Parameter Tuning

The following parameters model random noise in sensor measurements. More information on these parameters can be found in the “Inertial Sensor Noise Analysis Using Allan Variance” (Sensor Fusion and Tracking Toolbox) example.

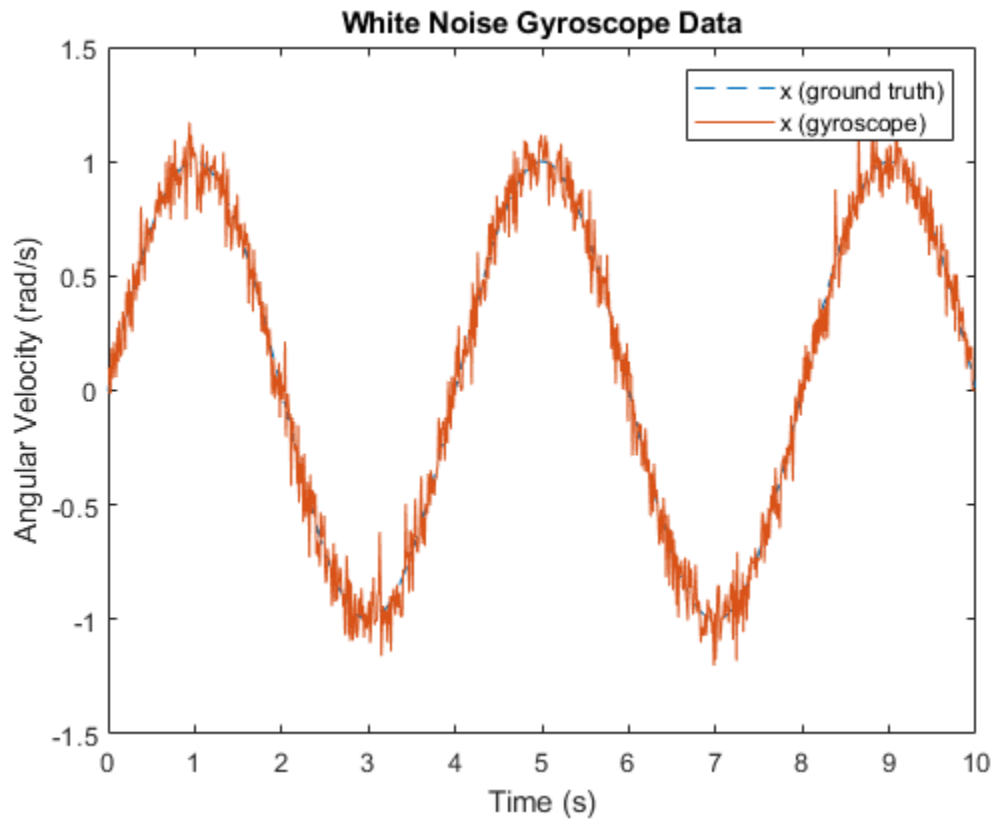
NoiseDensity is the amount of white noise in the sensor measurement. It is sometimes called angle random walk for gyroscopes or velocity random walk for accelerometers.

```
rng('default')

imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.NoiseDensity = 0.0125  ; % (rad/s)/sqrt(Hz)

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('White Noise Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

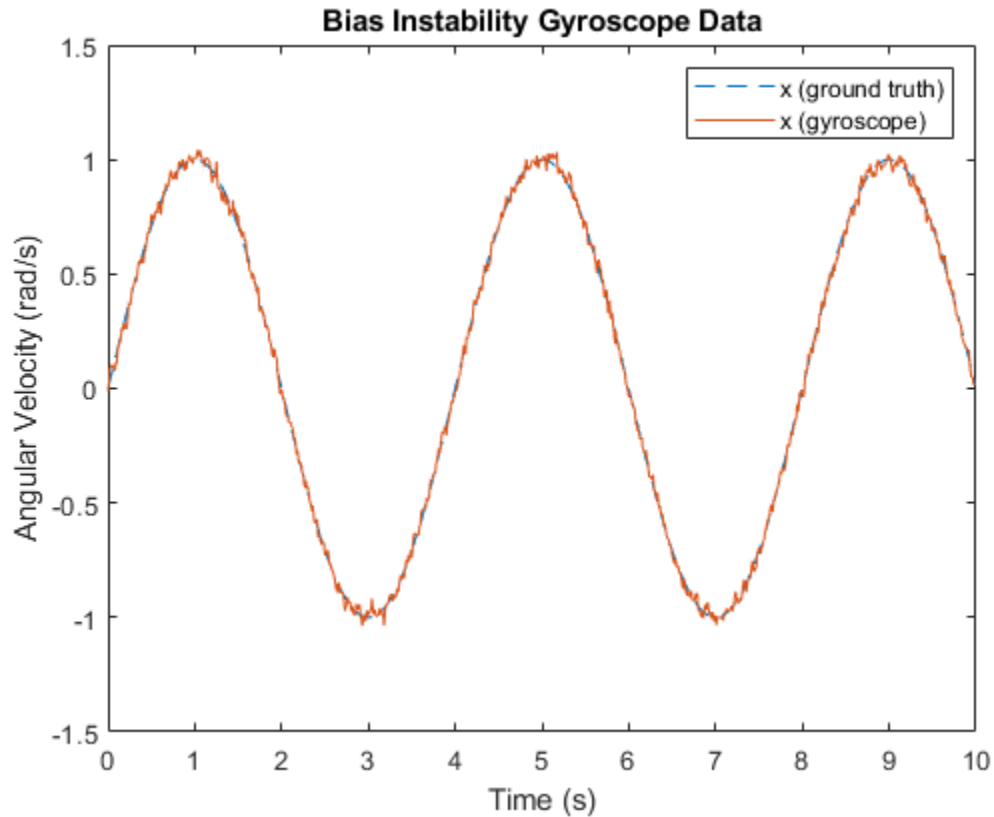


BiasInstability is the amount of pink or flicker noise in the sensor measurement.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.BiasInstability = 0.02  ; % rad/s

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Bias Instability Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

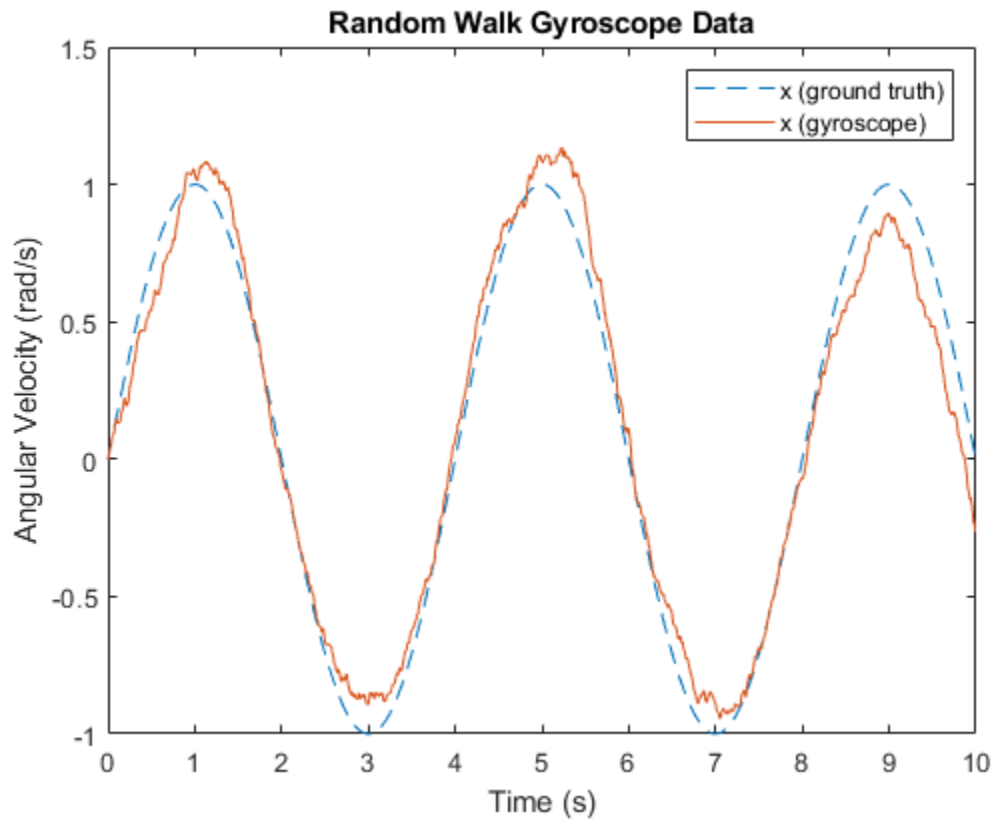



RandomWalk is the amount of Brownian noise in the sensor measurement. It is sometimes called rate random walk for gyroscopes or acceleration random walk for accelerometers.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.RandomWalk = 0.091  ; % (rad/s)*sqrt(Hz)

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Random Walk Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```



Environmental Parameter Tuning

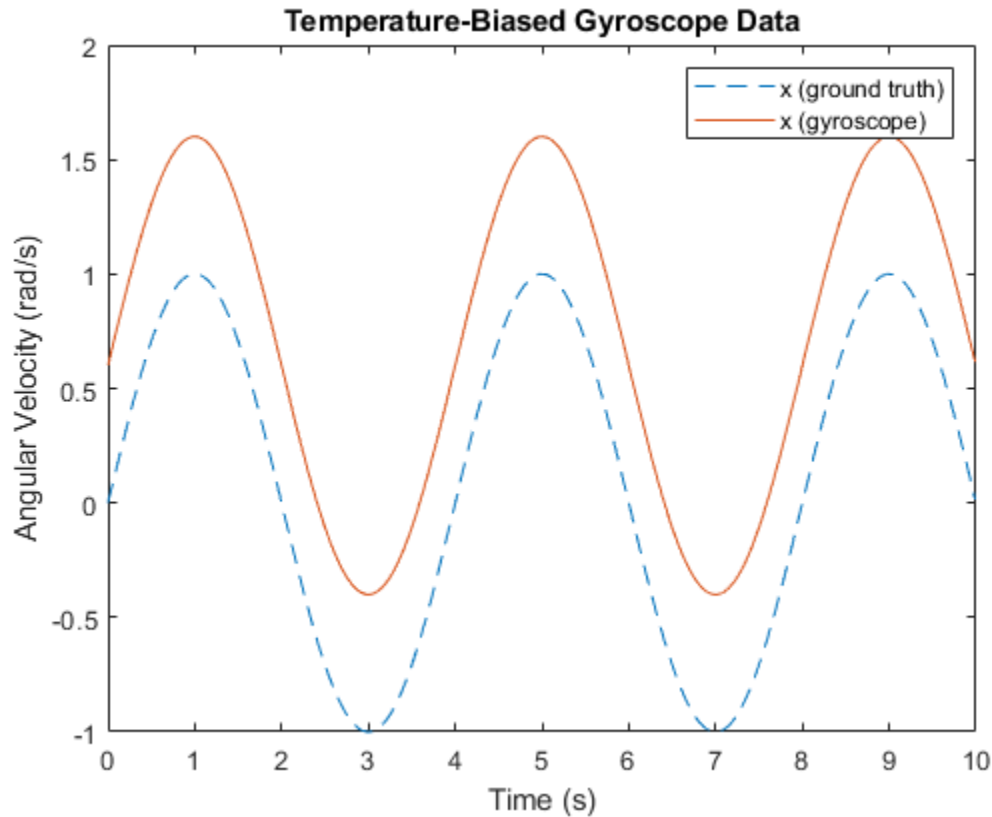
The following parameters model noise that arises from changes to the environment of the sensor.

TemperatureBias is the bias added to sensor measurements due to temperature difference from the default operating temperature. Most sensor datasheets list the default operating temperature as 25 degrees Celsius. This bias is shown by setting the parameter to a non-zero value and setting the operating temperature to a value above 25 degrees Celsius.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureBias = 0.06 ; % (rad/s)/(degrees C)
imu.Temperature = 35;

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Temperature-Biased Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```



TemperatureScaleFactor is the error in the sensor scale factor due to changes in the operating temperature. This causes errors in the scaling of the measurement; in other words smaller ideal values have less error than larger values. This error is shown by linearly increasing the temperature.

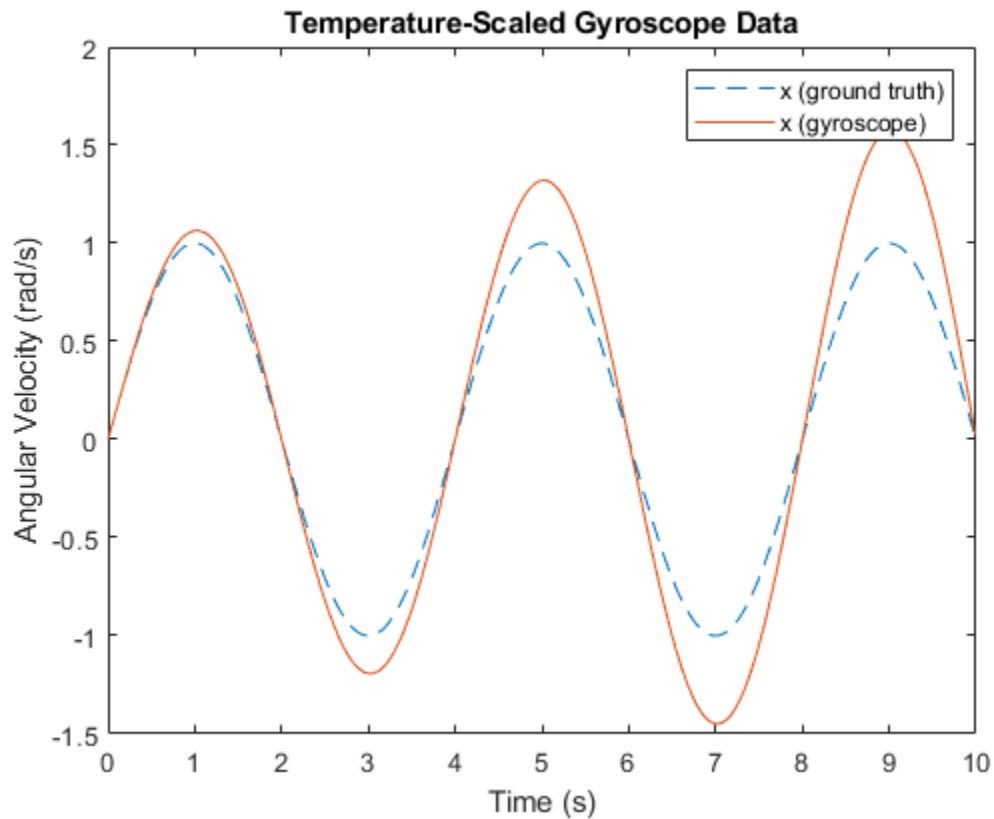
```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureScaleFactor = 3.2 ; % %/(degrees C)

standardTemperature = 25; % degrees C
temperatureSlope = 2; % (degrees C)/s

temperature = temperatureSlope*t + standardTemperature;

gyroData = zeros(N, 3);
for i = 1:N
    imu.Temperature = temperature(i);
    [~, gyroData(i,:)] = imu(acc(i,:), angvel(i,:));
end

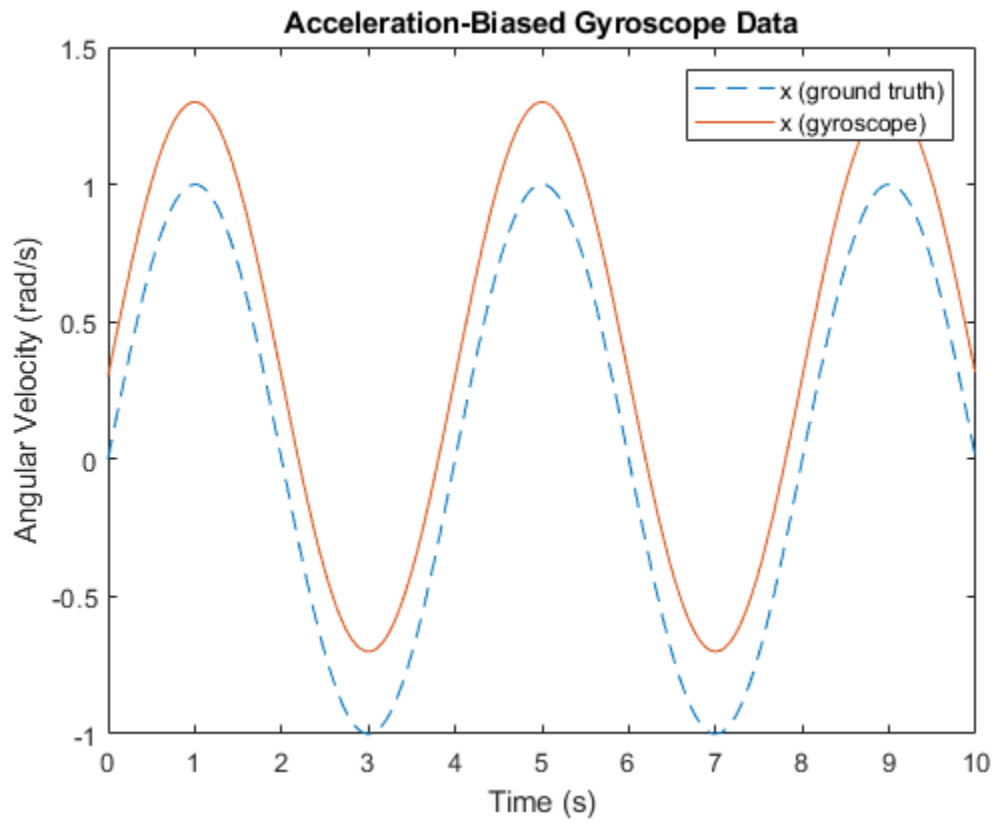
figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Temperature-Scaled Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```



AccelerationBias is the bias added to the gyroscope measurement due to linear accelerations. This parameter is specific to the gyroscope. This bias is shown by setting the parameter to a non-zero value and using a non-zero input acceleration.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.AccelerationBias = 0.3 ; % (rad/s)/(m/s^2)
acc(:,1) = 1;
[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Acceleration-Biased Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```



Logged Sensor Data Alignment for Orientation Estimation

This example shows how to align and preprocess logged sensor data. This allows the fusion filters to perform orientation estimation as expected. The logged data was collected from an accelerometer and a gyroscope mounted on a ground vehicle.

Load Logged Sensor Data

Load logged inertial measurement unit (IMU) data and extract individual sensor data and timestamps.

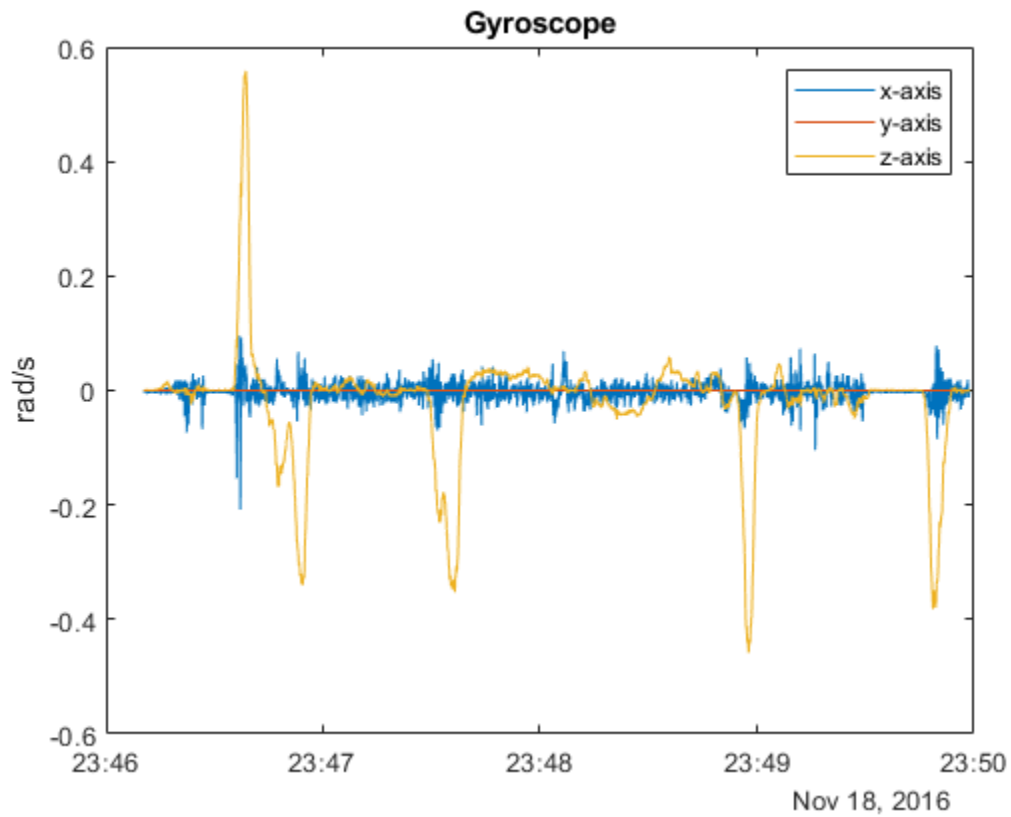
```
load('imuData', 'imuTT')

time = imuTT.Time;
accel = imuTT.LinearAcceleration;
gyro = imuTT.AngularVelocity;
orient = imuTT.Orientation;
```

Inspect the Gyroscope Data

From the range of angular velocity readings, the logged gyroscope data is in radians per second instead of degrees per second. Also, the larger z-axis values and small x- and y-axis values indicate that the device rotated around the z-axis only.

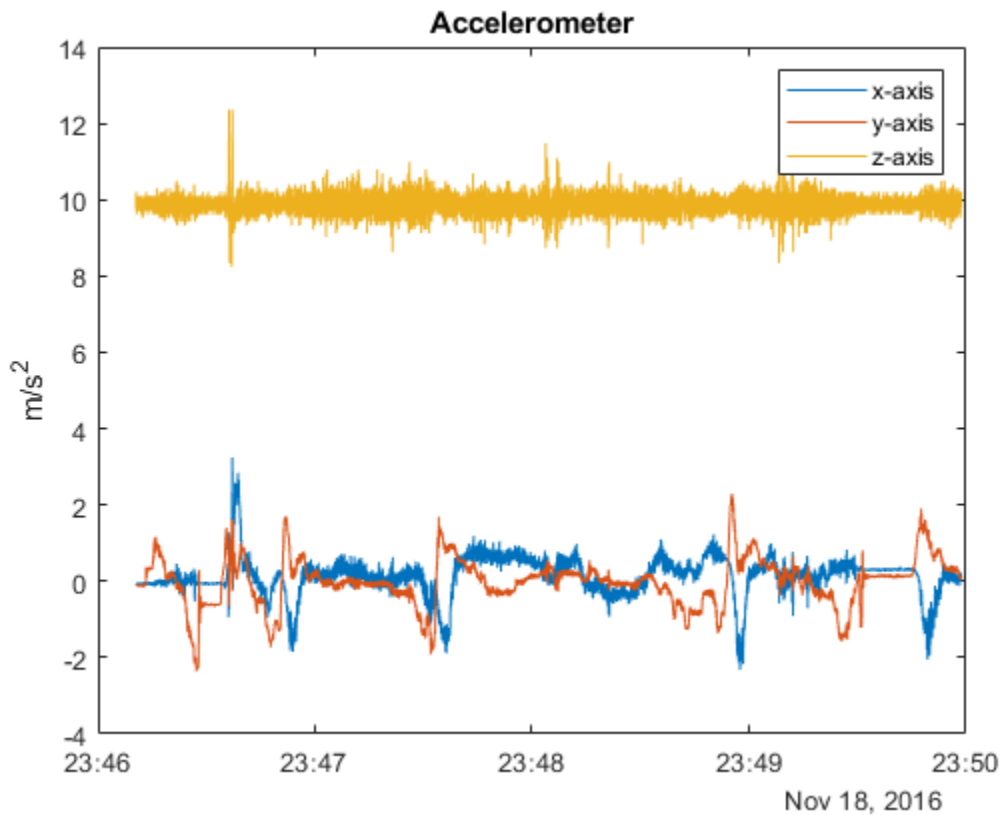
```
figure
plot(time, gyro)
title('Gyroscope')
ylabel('rad/s')
legend('x-axis', 'y-axis', 'z-axis')
```



Inspect the Accelerometer Data

Since the z-axis reading of the accelerometer is around 10, the logged data is in meters per second squared instead of g's.

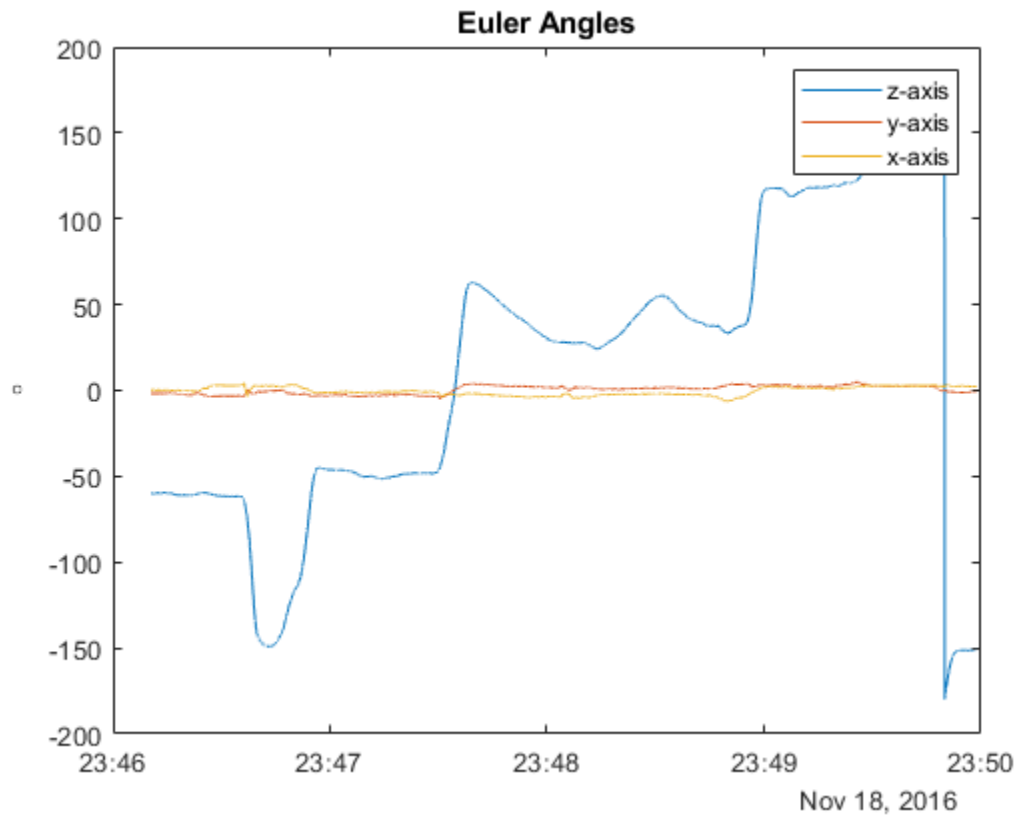
```
figure
plot(time, accel)
title('Accelerometer')
ylabel('m/s^2')
legend('x-axis', 'y-axis', 'z-axis')
```



Inspect the Orientation Data

Convert the logged orientation quaternion data to Euler angles in degrees. The z-axis is changing while the x- and y-axis are relatively fixed. This matches the gyroscope and accelerometer readings. Therefore, no axis negating or rotating is required. However, the z-axis Euler angle is decreasing while the gyroscope reading is positive. This means that the logged orientation quaternion is expected to be applied as a point rotation operator ($v' = qvq^*$). In order to have the orientation quaternion match the orientations filters, such as `imufilter`, the quaternion needs to be applied as a frame rotation operator ($v' = q^*vq$). This can be done by conjugating the logged orientation quaternion.

```
figure
plot(time, eulerd(orient, 'ZYX', 'frame'))
title('Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
```

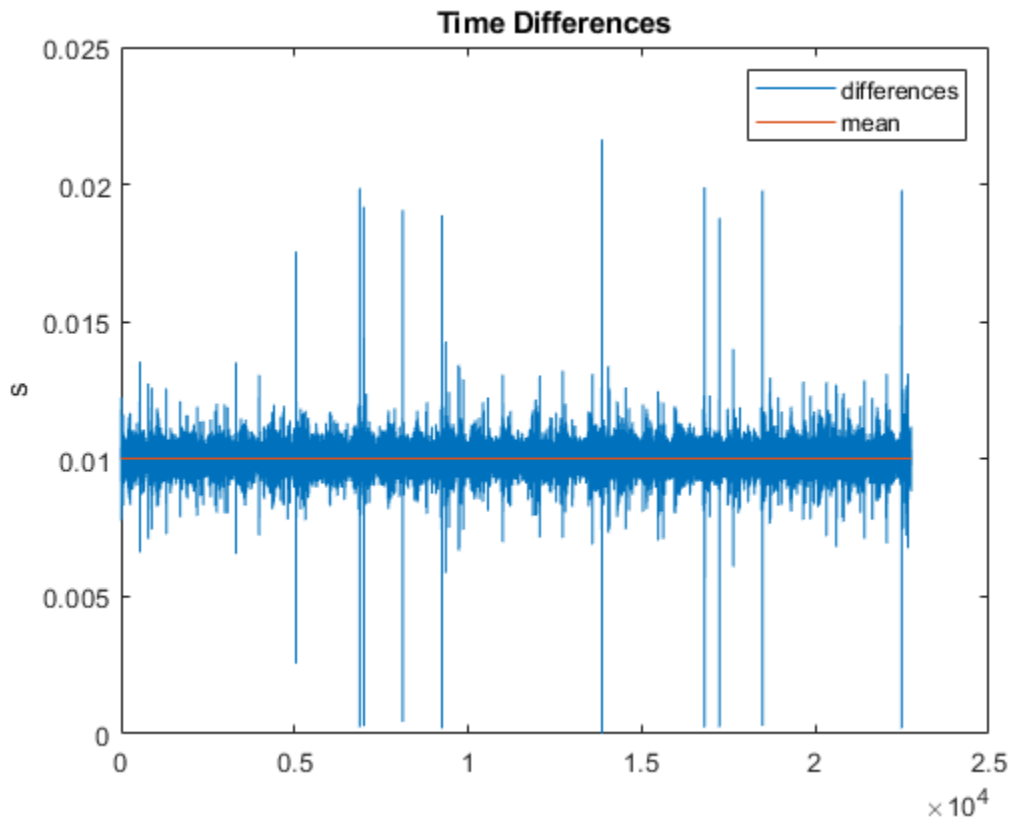



Find the Sampling Rate of the Logged Data

An estimate of the sampling rate can be obtained by taking the mean of the difference between the timestamps. Notice that there are some variances in the time differences. Since the variances are small for this logged data, the mean of the time differences can be used. Alternatively, the sensor data could be interpolated using the timestamps and equally spaced timestamps as query points.

```
deltaTimes = seconds(diff(time));
sampleRate = 1/mean(deltaTimes);
```

```
figure
plot([deltaTimes, repmat(mean(deltaTimes), numel(deltaTimes), 1)])
title('Time Differences')
ylabel('s')
legend('differences', 'mean')
```



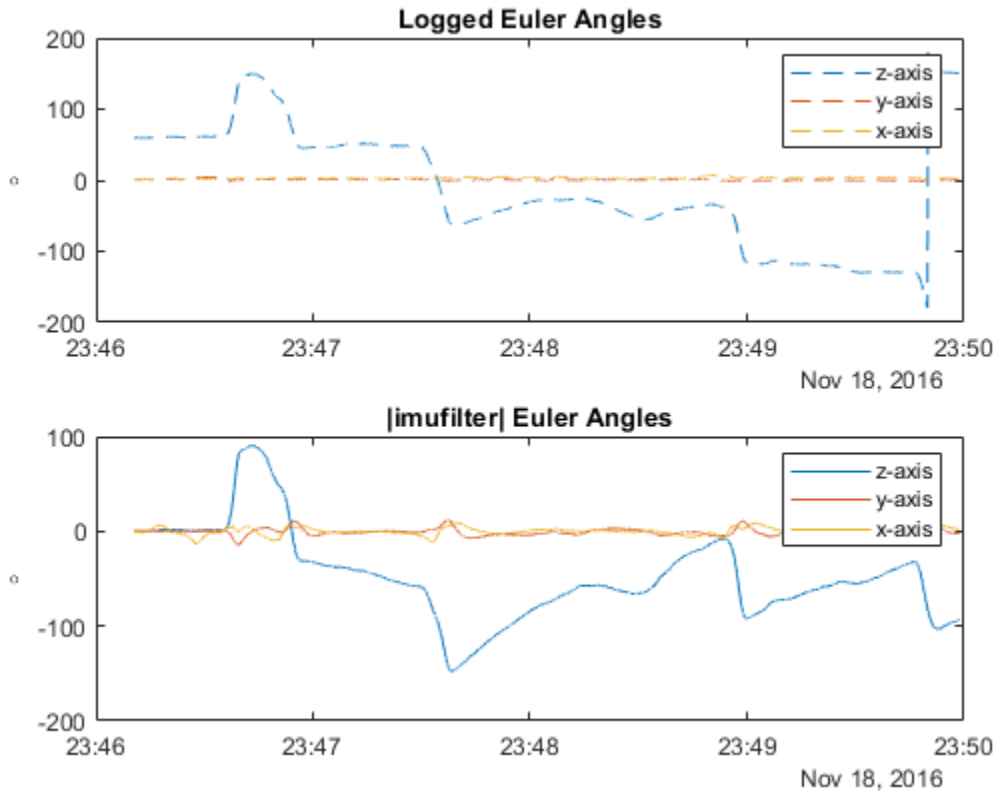
Compare the Transformed Logged Quaternion to the `imufilter` Quaternion

Conjugate the logged orientation quaternion before comparing it to the estimated orientation quaternion from `imufilter`. From the plot below, there is still a constant offset in the z-axis Euler angle estimate. This is because the `imufilter` assumes the initial orientation of the device is aligned with the navigation frame.

```
loggedOrient = conj(orient);

filt = imufilter('SampleRate', sampleRate);
estOrient = filt(accel, gyro);

figure
subplot(2, 1, 1)
plot(time, eulerd(loggedOrient, 'ZYX', 'frame'), '--')
title('Logged Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
subplot(2, 1, 2)
plot(time, eulerd(estOrient, 'ZYX', 'frame'))
title('|imufilter| Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
```

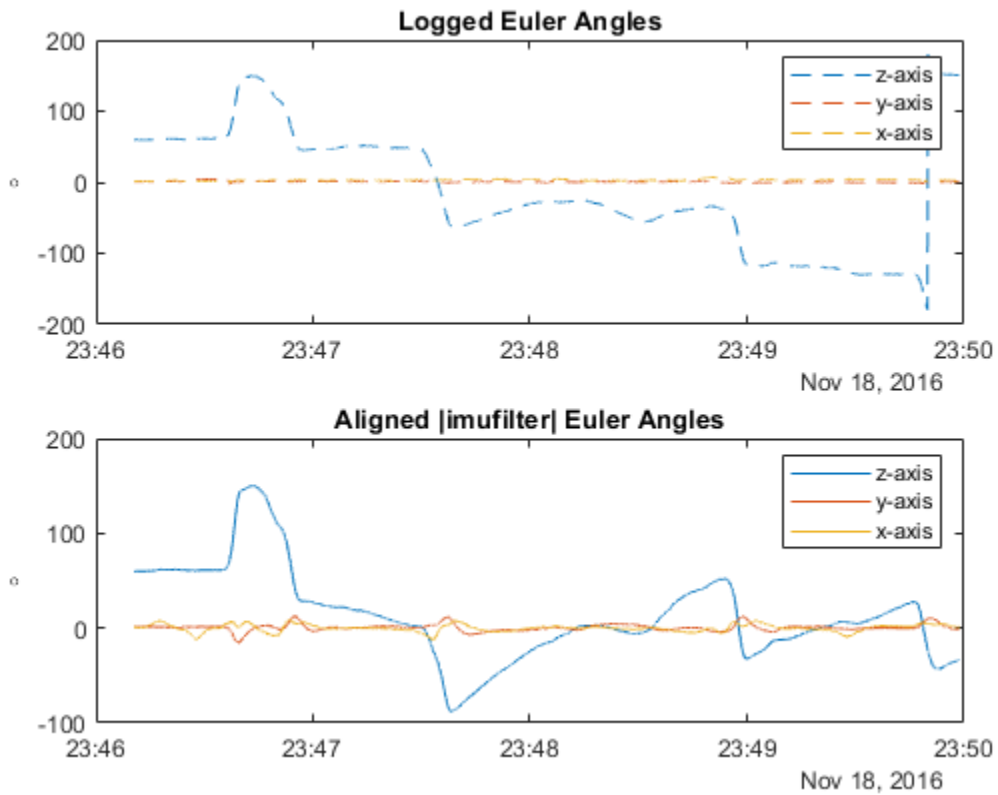


Align the Logged Orientation and imufilter Orientation

Align the `imufilter` orientation quaternion with the logged orientation quaternion by applying a constant bias using the first logged orientation quaternion. For quaternions, a constant rotation bias can be applied by pre-multiplying frame rotations or post-multiplying point rotations. Since `imufilter` reports quaternions as frame rotation operators, the estimated orientation quaternions are pre-multiplied by the first logged orientation quaternion.

```
alignedEstOrient = loggedOrient(1) .* estOrient;
```

```
figure
subplot(2, 1, 1)
plot(time, eulerd(loggedOrient, 'ZYX', 'frame'), '--')
title('Logged Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
subplot(2, 1, 2)
plot(time, eulerd(alignedEstOrient, 'ZYX', 'frame'))
title('Aligned |imufilter| Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
```



Conclusion

For the MAT-file in this example, you checked the following aspects for alignment:

- Units for accelerometer and gyroscope.
- Axes alignments of accelerometer and gyroscope.
- Orientation quaternion rotation operator (point: $v' = qvq^*$ or frame: $v' = q^*vq$)

Different unit conversions, axes alignments, and quaternion transformations may need to be applied depending on the format of the logged data.

Generate Off-Centered IMU Readings

This example shows how to generate inertial measurement unit (IMU) readings from a sensor that is mounted on a ground vehicle. Depending on the location of the sensor, the IMU accelerations are different.

Create Trajectory

Specify the waypoint trajectory of a vehicle and compute the vehicle poses using `lookupPose`.

```
% Sampling rate.
Fs = 100;

% Waypoints and times of arrival.
waypoints = [1 1 1; 3 1 1; 3 0 0; 0 0 0];
t = [1; 10; 20; 30];

% Create trajectory and compute pose.
traj = waypointTrajectory(waypoints, t, "SampleRate", Fs);
[posVeh, orientVeh, velVeh, accVeh, angvelVeh] = lookupPose(traj, ...
    t(1):1/Fs:t(end));
```

Create Sensor and Define Offset

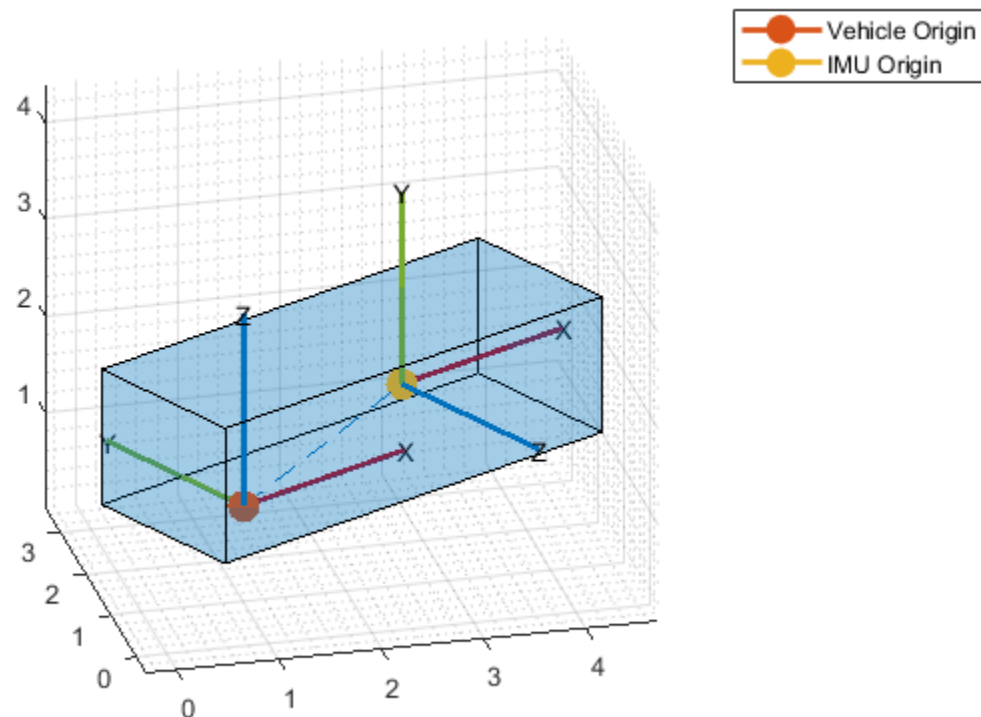
Create two 9-axis `imuSensor` objects composed of accelerometer, gyroscope, and magnetometer sensors. One `imuSensor` object generates readings of an IMU mounted at the vehicle's origin and the other one generates readings of an IMU mounted at the driver's seat. Next, specify the offset between the vehicle origin and the IMU mounted at the driver's seat. Call `helperPlotIMU` to visualize the locations of the sensors.

```
% IMU at vehicle origin.
imu = imuSensor("accel-gyro-mag", "SampleRate", Fs);

% IMU at driver's seat.
mountedIMU = imuSensor("accel-gyro-mag", "SampleRate", Fs);

% Position and orientation offset of the vehicle and the mounted IMU.
posVeh2IMU = [2.4 0.5 0.4];
orientVeh2IMU = quaternion([0 0 90], "eulerd", "ZYX", "frame");

% Visualization.
helperPlotIMU(posVeh(1,:), orientVeh(1,:), posVeh2IMU, orientVeh2IMU);
```



Calculate IMU Trajectory Using Vehicle Trajectory

Compute the ground truth trajectory of the IMU mounted at the driver's seat using the `transformMotion` function. This function uses the position and orientation offsets and the vehicle trajectory to compute the IMU trajectory.

```
[posIMU, orientIMU, velIMU, accIMU, angvelIMU] = transformMotion( ...
    posVeh2IMU, orientVeh2IMU, ...
    posVeh, orientVeh, velVeh, accVeh, angvelVeh);
```

Generate Sensor Readings

Generate the IMU readings for both the IMU mounted at the vehicle origin and the IMU mounted at the driver's seat.

```
% IMU at vehicle origin.
[accel, gyro, mag] = imu(accVeh, angvelVeh, orientVeh);

% IMU at driver's seat.
[accelMounted, gyroMounted, magMounted] = mountedIMU( ...
    accIMU, angvelIMU, orientIMU);
```

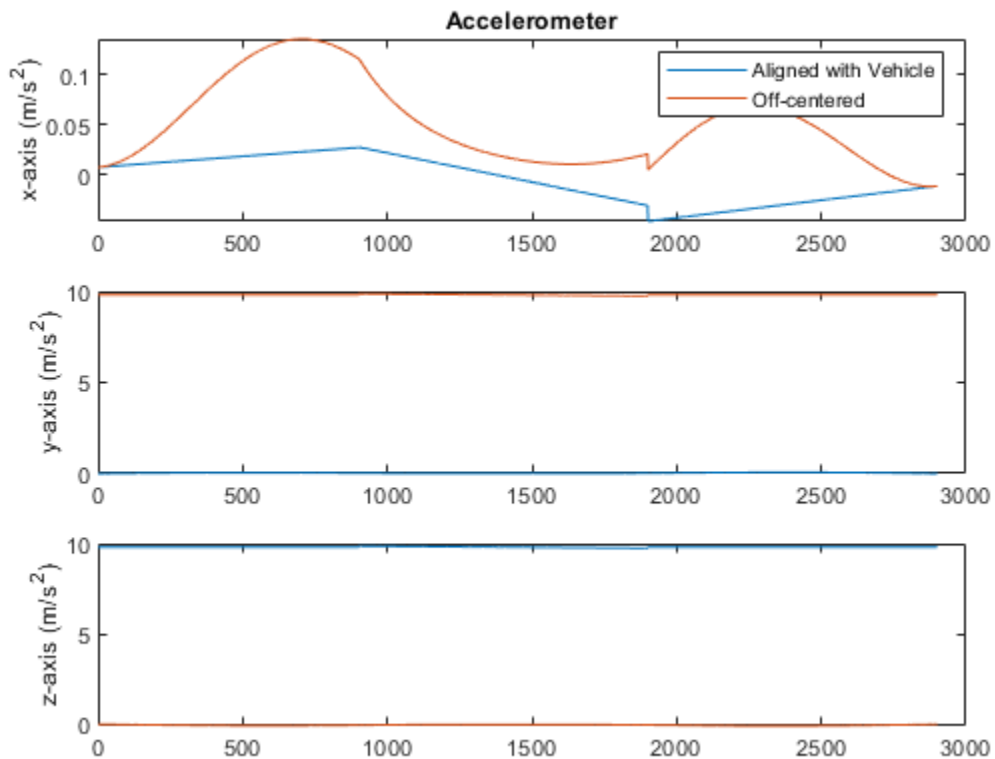
Compare Accelerometer Readings

Compare the accelerometer readings of the two IMUs. Notice that the x-axis acceleration is different because of the off-center location.

```

figure('Name', 'Accelerometer Comparison')
subplot(3, 1, 1)
plot([accel(:,1), accelMounted(:,1)])
legend('Aligned with Vehicle', 'Off-centered')
title('Accelerometer')
ylabel('x-axis (m/s^2)')
subplot(3, 1, 2)
plot([accel(:,2), accelMounted(:,2)])
ylabel('y-axis (m/s^2)')
subplot(3, 1, 3)
plot([accel(:,3), accelMounted(:,3)])
ylabel('z-axis (m/s^2)')

```



Compare Gyroscope Readings

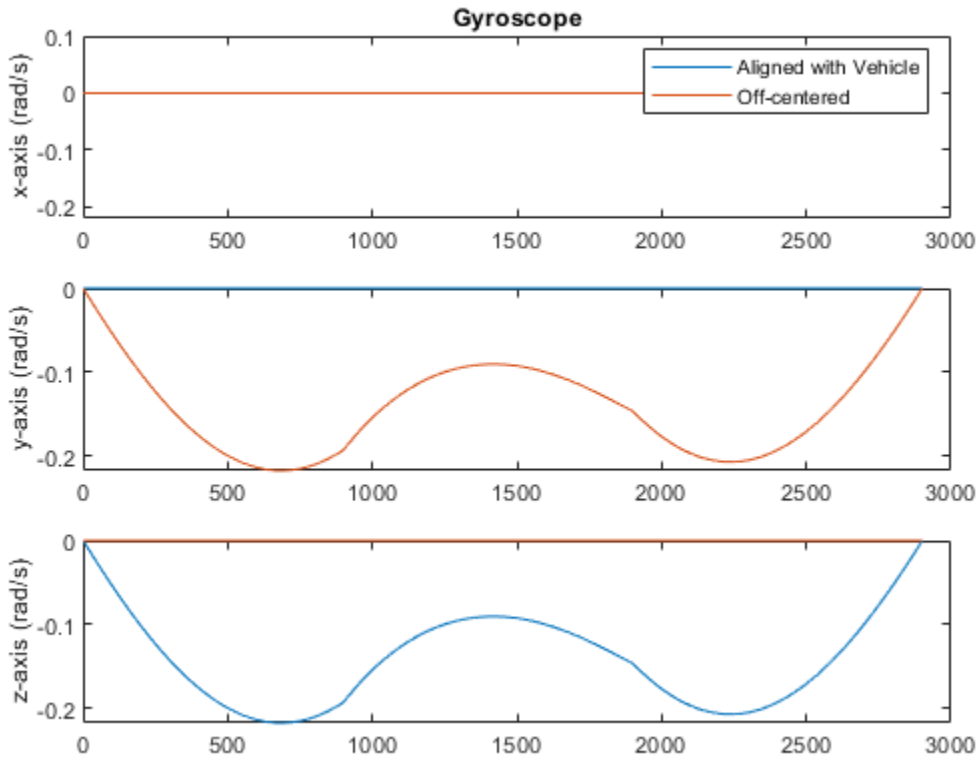
Compare the gyroscope readings of the two IMUs.

```

figure('Name', 'Gyroscope Comparison')
subplot(3, 1, 1)
plot([gyro(:,1), gyroMounted(:,1)])
ylim([-0.22 0.1])
legend('Aligned with Vehicle', 'Off-centered')
title('Gyroscope')
ylabel('x-axis (rad/s)')
subplot(3, 1, 2)
plot([gyro(:,2), gyroMounted(:,2)])
ylabel('y-axis (rad/s)')
subplot(3, 1, 3)

```

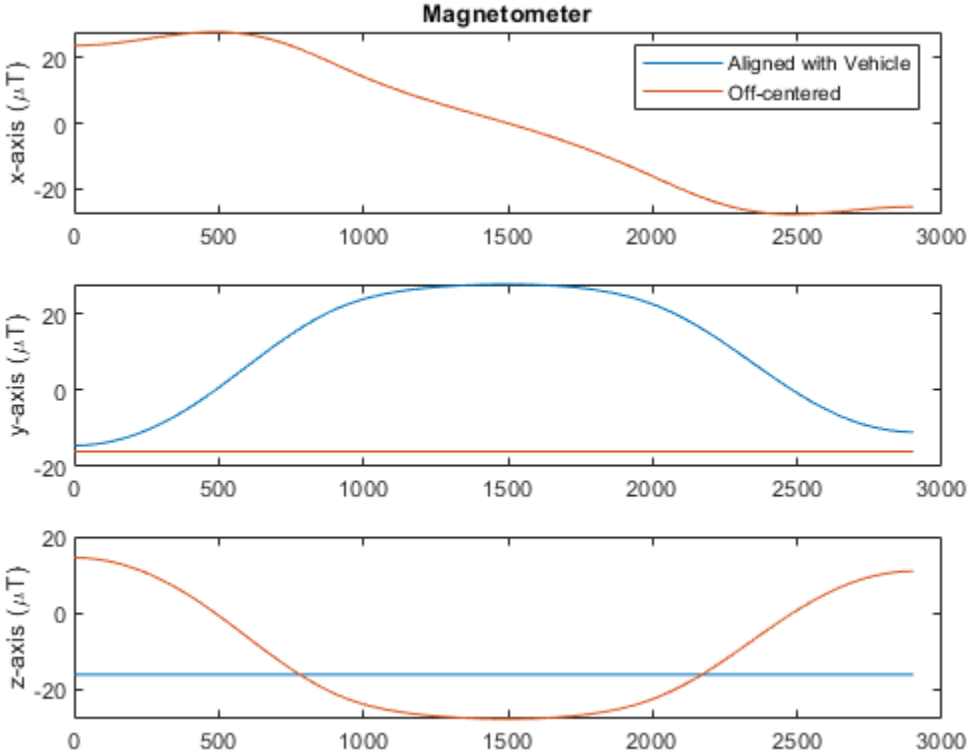
```
plot([gyro(:,3), gyroMounted(:,3)])
ylabel('z-axis (rad/s)')
```



Compare Magnetometer Readings

Compare the magnetometer readings of the two IMUs.

```
figure('Name', 'Magnetometer Comparison')
subplot(3, 1, 1)
plot([mag(:,1), magMounted(:,1)])
legend('Aligned with Vehicle', 'Off-centered')
title('Magnetometer')
ylabel('x-axis (\muT)')
subplot(3, 1, 2)
plot([mag(:,2), magMounted(:,2)])
ylabel('y-axis (\muT)')
subplot(3, 1, 3)
plot([mag(:,3), magMounted(:,3)])
ylabel('z-axis (\muT)')
```

Estimate Robot Pose with Scan Matching

This example demonstrates how to match two laser scans using the Normal Distributions Transform (NDT) algorithm [1]. The goal of scan matching is to find the relative pose (or transform) between the two robot positions where the scans were taken. The scans can be aligned based on the shapes of their overlapping features.

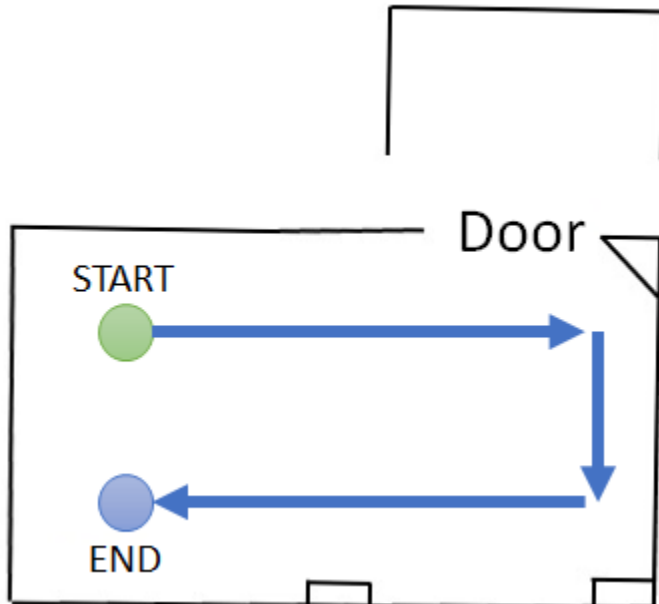
To estimate this pose, NDT subdivides the laser scan into 2D cells and each cell is assigned a corresponding normal distribution. The distribution represents the probability of measuring a point in that cell. Once the probability density is calculated, an optimization method finds the relative pose between the current laser scan and the reference laser scan. To speed up the convergence of the method, an initial guess of the pose can be provided. Typically, robot odometry is used to supply the initial estimate.

If you apply scan matching to a sequence of scans, you can use it to recover a rough map of the environment that the robot traverses. Scan matching also plays a crucial role in other applications, such as position tracking and Simultaneous Localization and Mapping (SLAM).

Load Laser Scan Data from File

```
load lidarScans.mat
```

The laser scan data was collected by a mobile robot in an indoor environment. An approximate floorplan of the area, along with the robot's path through the space, is shown in the following image.



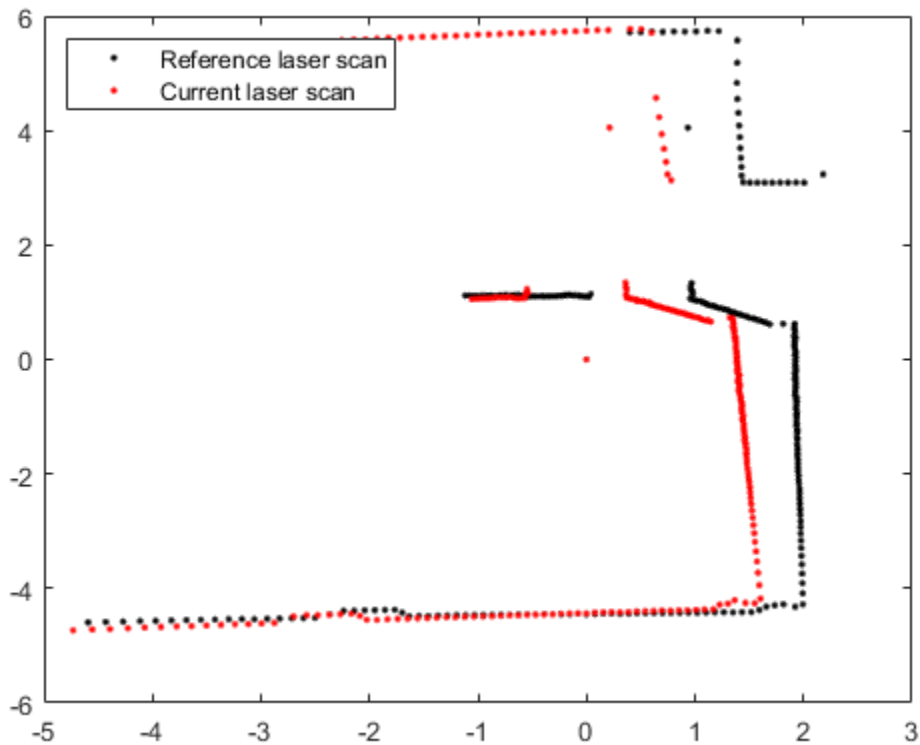
Plot Two Laser Scans

Pick two laser scans to scan match from `lidarScans`. They should share common features by being close together in the sequence.

```
referenceScan = lidarScans(180);  
currentScan = lidarScans(202);
```

Display the two scans. Notice there are translational and rotational offsets, but some features still match.

```
currScanCart = currentScan.Cartesian;
refScanCart = referenceScan.Cartesian;
figure
plot(refScanCart(:,1),refScanCart(:,2),'k. ');
hold on
plot(currScanCart(:,1),currScanCart(:,2),'r. ');
legend('Reference laser scan','Current laser scan','Location','NorthWest');
```



Run Scan Matching Algorithm and Display Transformed Scan

Pass these two scans to the scan matching function. `matchScans` calculates the relative pose of the current scan with respect to the reference scan.

```
transform = matchScans(currentScan,referenceScan)
```

```
transform = 1x3
```

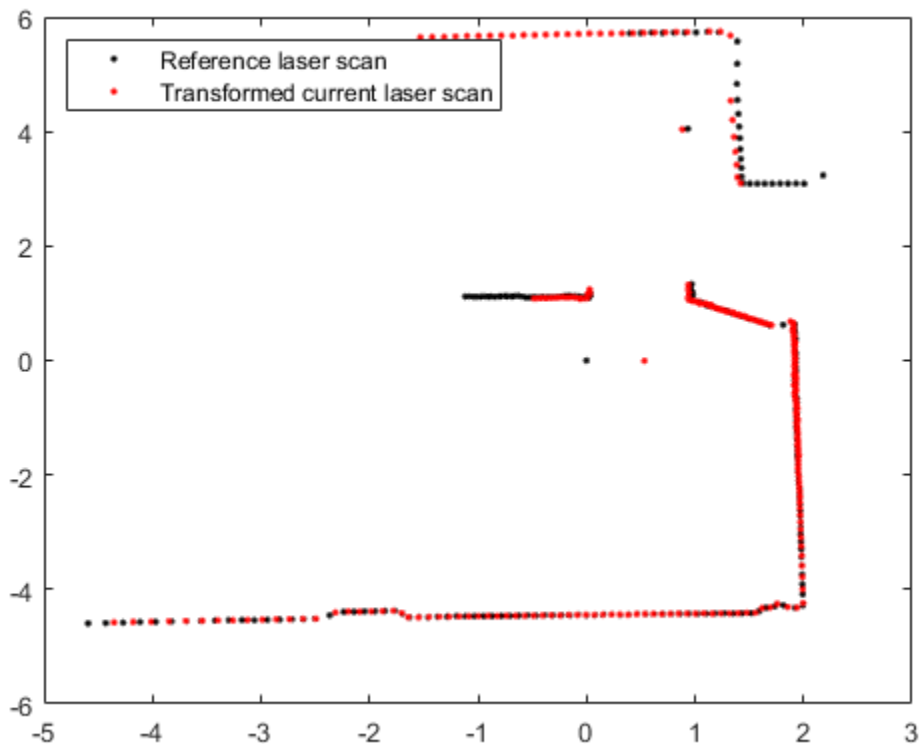
```
    0.5348    -0.0065    -0.0336
```

To visually verify that the relative pose was calculated correctly, transform the current scan by the calculated pose using `transformScan`. This transformed laser scan can be used to visualize the result.

```
transScan = transformScan(currentScan,transform);
```

Display the reference scan alongside the transformed current laser scan. If the scan matching was successful, the two scans should be well-aligned.

```
figure
plot(refScanCart(:,1),refScanCart(:,2),'k. ');
hold on
transScanCart = transScan.Cartesian;
plot(transScanCart(:,1),transScanCart(:,2),'r. ');
legend('Reference laser scan','Transformed current laser scan','Location','NorthWest');
```



Build Occupancy Grid Map Using Iterative Scan Matching

If you apply scan matching to a sequence of scans, you can use it to recover a rough map of the environment. Use the `occupancyMap` class to build a probabilistic occupancy grid map of the environment.

Create an occupancy grid object for a 15 meter by 15 meter area. Set the map's origin to be [-7.5 -7.5].

```
map = occupancyMap(15,15,20);
map.GridLocationInWorld = [-7.5 -7.5]
```

```
map =
  occupancyMap with properties:
    mapLayer Properties
      OccupiedThreshold: 0.6500
      FreeThreshold: 0.2000
```

```

ProbabilitySaturation: [0.0010 0.9990]
    LayerName: 'probabilityLayer'
    DataType: 'double'
    DefaultValue: 0.5000
GridLocationInWorld: [-7.5000 -7.5000]
  GridOriginInLocal: [0 0]
  LocalOriginInWorld: [-7.5000 -7.5000]
    Resolution: 20
    GridSize: [300 300]
  XLocalLimits: [0 15]
  YLocalLimits: [0 15]
  XWorldLimits: [-7.5000 7.5000]
  YWorldLimits: [-7.5000 7.5000]

```

Pre-allocate an array to capture the absolute movement of the robot. Initialize the first pose as $[0 \ 0 \ 0]$. All other poses are relative to the first measured scan.

```

numScans = numel(lidarScans);
initialPose = [0 0 0];
poseList = zeros(numScans,3);
poseList(1,:) = initialPose;
transform = initialPose;

```

Create a loop for processing the scans and mapping the area. The laser scans are processed in pairs. Define the first scan as reference scan and the second scan as current scan. The two scans are then passed to the scan matching algorithm and the relative pose between the two scans is computed. The `exampleHelperComposeTransform` function is used to calculate of the cumulative absolute robot pose. The scan data along with the absolute robot pose can then be passed into the `insertRay` function of the occupancy grid.

```

% Loop through all the scans and calculate the relative poses between them
for idx = 2:numScans
    % Process the data in pairs.
    referenceScan = lidarScans(idx-1);
    currentScan = lidarScans(idx);

    % Run scan matching. Note that the scan angles stay the same and do
    % not have to be recomputed. To increase accuracy, set the maximum
    % number of iterations to 500. Use the transform from the last
    % iteration as the initial estimate.
    [transform,stats] = matchScans(currentScan,referenceScan, ...
        'MaxIterations',500,'InitialPose',transform);

    % The |Score| in the statistics structure is a good indication of the
    % quality of the scan match.
    if stats.Score / currentScan.Count < 1.0
        disp(['Low scan match score for index ' num2str(idx) '. Score = ' num2str(stats.Score) '
        end

    % Maintain the list of robot poses.
    absolutePose = exampleHelperComposeTransform(poseList(idx-1,:),transform);
    poseList(idx,:) = absolutePose;

    % Integrate the current laser scan into the probabilistic occupancy
    % grid.
    insertRay(map,absolutePose,currentScan,10);

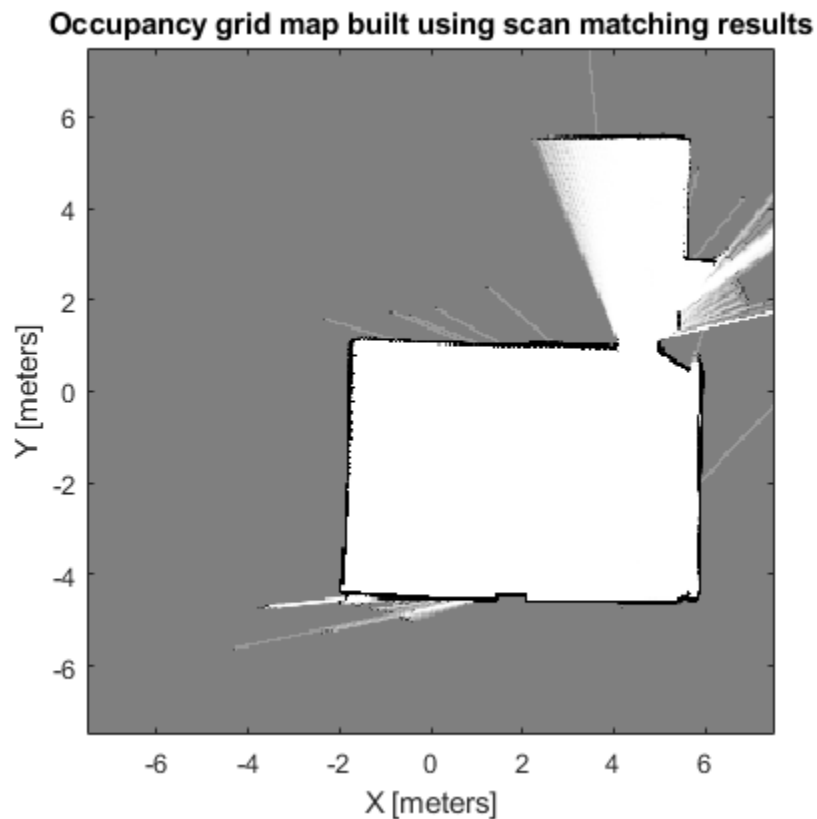
```

```
end
```

Visualize Map

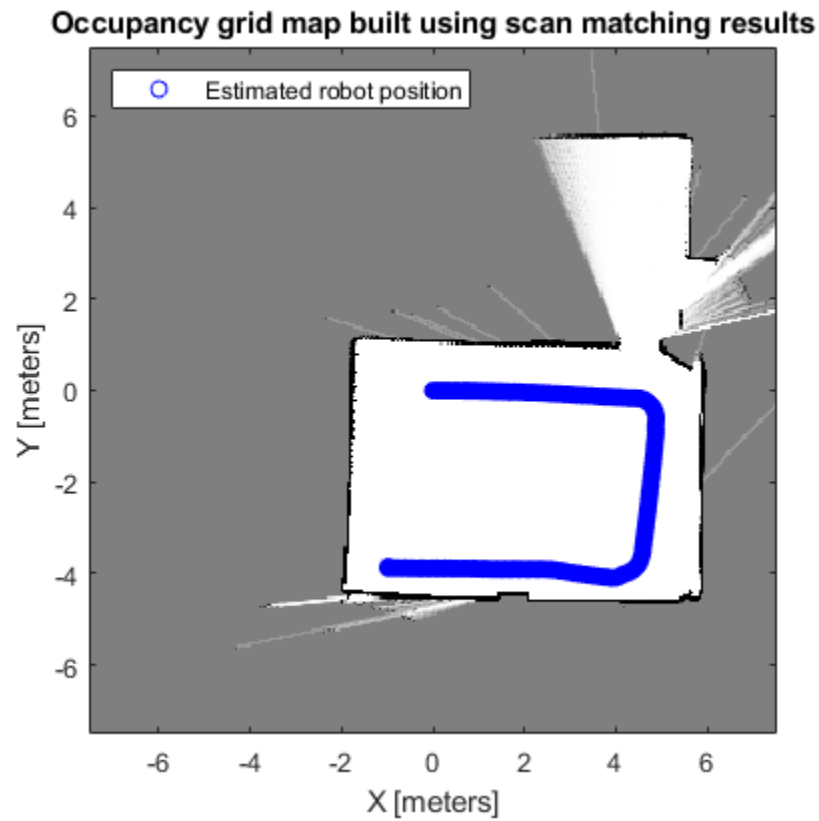
Visualize the occupancy grid map populated with the laser scans.

```
figure  
show(map);  
title('Occupancy grid map built using scan matching results');
```



Plot the absolute robot poses that were calculated by the scan matching algorithm. This shows the path that the robot took through the map of the environment.

```
hold on  
plot(poseList(:,1),poseList(:,2),'bo','DisplayName','Estimated robot position');  
legend('show','Location','NorthWest')
```



References

- [1] P. Biber, W. Strasser, "The normal distributions transform: A new approach to laser scan matching," in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2003, pp. 2743-2748

Localize TurtleBot Using Monte Carlo Localization

This example demonstrates an application of the Monte Carlo Localization (MCL) algorithm on TurtleBot® in simulated Gazebo® environment.

Monte Carlo Localization (MCL) is an algorithm to localize a robot using a particle filter. The algorithm requires a known map and the task is to estimate the pose (position and orientation) of the robot within the map based on the motion and sensing of the robot. The algorithm starts with an initial belief of the robot pose's probability distribution, which is represented by particles distributed according to such belief. These particles are propagated following the robot's motion model each time the robot's pose changes. Upon receiving new sensor readings, each particle will evaluate its accuracy by checking how likely it would receive such sensor readings at its current pose. Next the algorithm will redistribute (resample) particles to bias particles that are more accurate. Keep iterating these moving, sensing and resampling steps, and all particles should converge to a single cluster near the true pose of robot if localization is successful.

Adaptive Monte Carlo Localization (AMCL) is the variant of MCL implemented in `monteCarloLocalization`. AMCL dynamically adjusts the number of particles based on KL-distance [1] to ensure that the particle distribution converge to the true distribution of robot state based on all past sensor and motion measurements with high probability.

The current MATLAB® AMCL implementation can be applied to any differential drive robot equipped with a range finder.

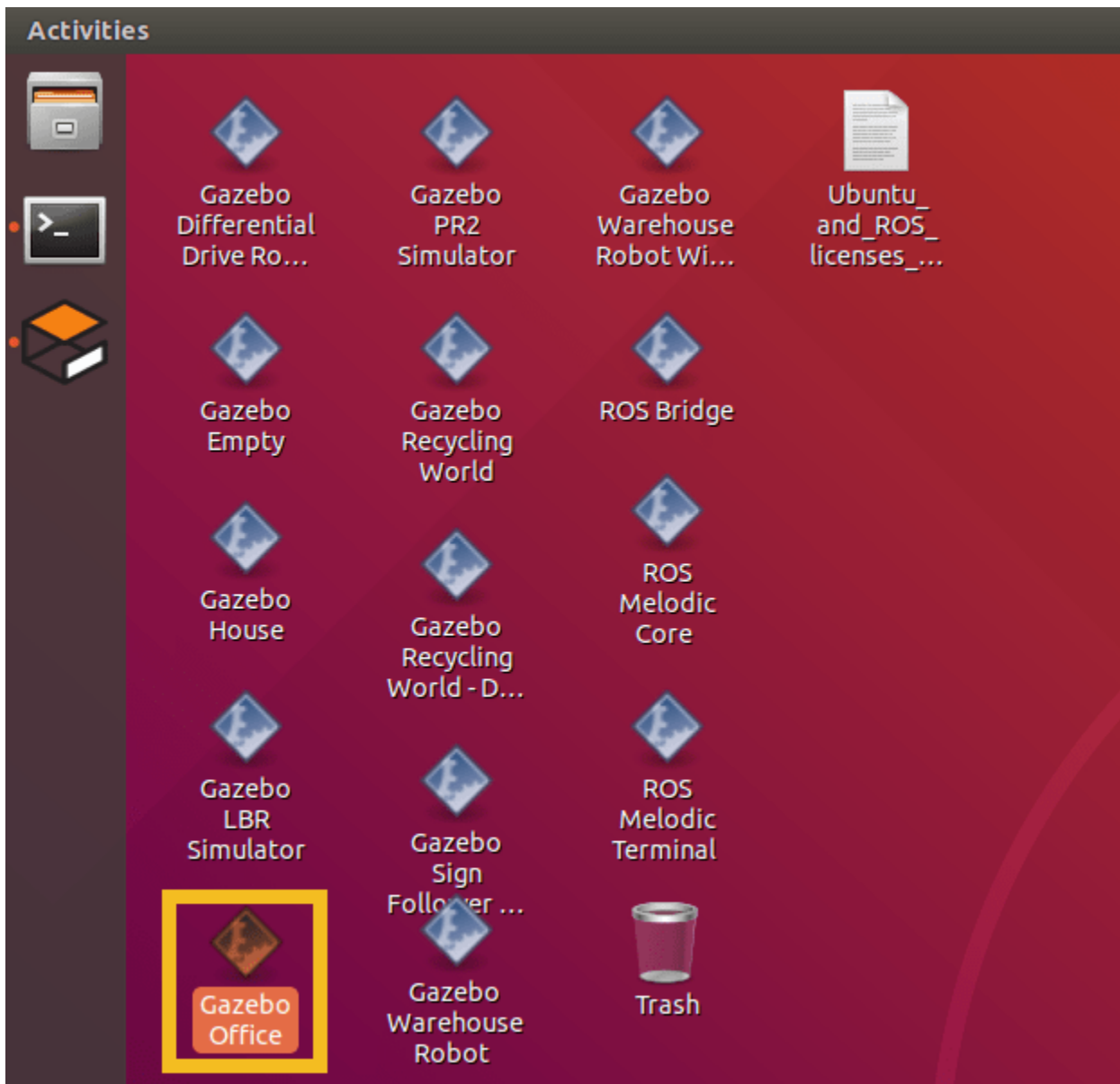
The Gazebo TurtleBot simulation must be running for this example to work.

Prerequisites: “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox), “Access the tf Transformation Tree in ROS” (ROS Toolbox), “Exchange Data with ROS Publishers and Subscribers” (ROS Toolbox).

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the `System` object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

Connect to the TurtleBot in Gazebo

First, spawn a simulated TurtleBot inside an office environment in a virtual machine by following steps in the “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox) to launch the **Gazebo Office World** from the desktop, as shown below.

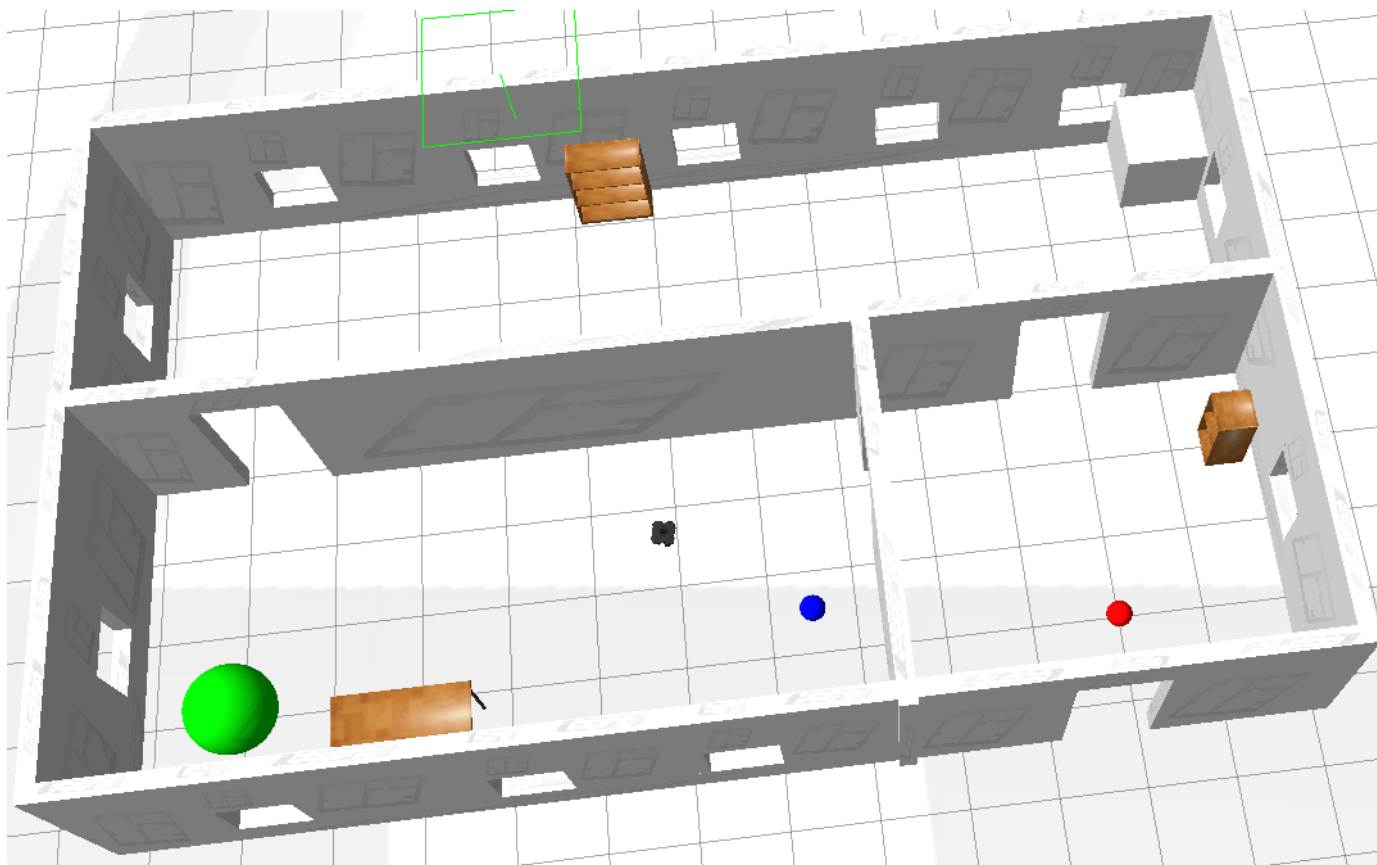


In your MATLAB instance on the host computer, run the following commands to initialize ROS global node in MATLAB and connect to the ROS master in the virtual machine through its IP address `ipaddress`. Replace `ipaddress` with the IP address of your TurtleBot in virtual machine.

```
ipaddress = '192.168.2.150';
rosinit(ipaddress,11311);
```

Initializing global node /matlab_global_node_73841 with NodeURI http://192.168.2.1:53461/

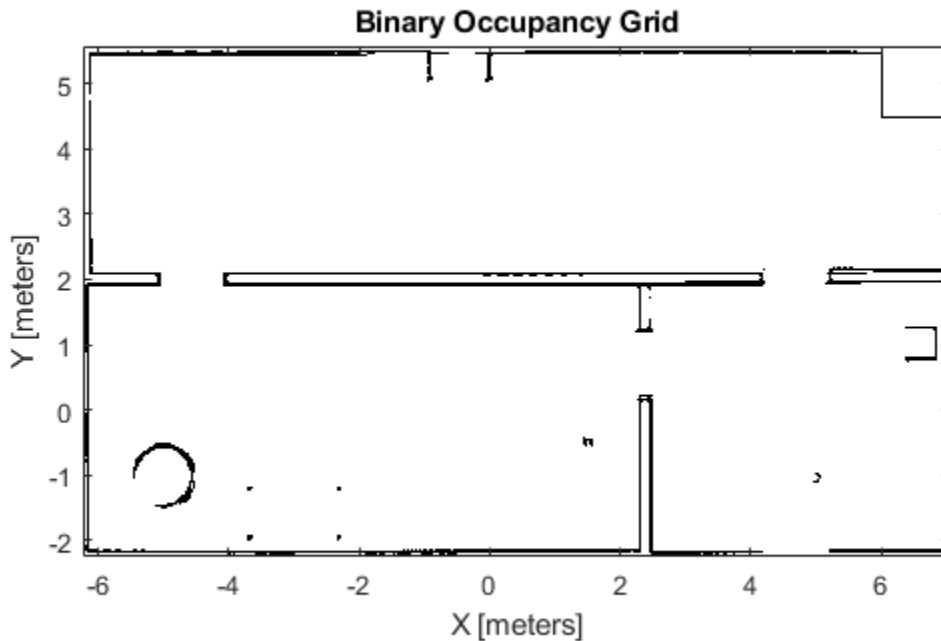
The layout of simulated office environment:



Load the Map of the Simulation World

Load a binary occupancy grid of the office environment in Gazebo. The map is generated by driving TurtleBot inside the office environment. The map is constructed using range-bearing readings from Kinect® and ground truth poses from `gazebo/model_states` topic.

```
load officemap.mat  
show(map)
```



Setup the Laser Sensor Model and TurtleBot Motion Model

TurtleBot can be modeled as a differential drive robot and its motion can be estimated using odometry data. The `Noise` property defines the uncertainty in robot's rotational and linear motion. Increasing the `odometryModel.Noise` property will allow more spread when propagating particles using odometry measurements. Refer to `odometryMotionModel` for property details.

```
odometryModel = odometryMotionModel;
odometryModel.Noise = [0.2 0.2 0.2 0.2];
```

The sensor on TurtleBot is a simulated range finder converted from Kinect readings. The likelihood field method is used to compute the probability of perceiving a set of measurements by comparing the end points of the range finder measurements to the occupancy map. If the end points match the occupied points in occupancy map, the probability of perceiving such measurements is high. The sensor model should be tuned to match the actual sensor property to achieve better test results. The property `SensorLimits` defines the minimum and maximum range of sensor readings. The property `Map` defines the occupancy map used for computing likelihood field. Please refer to `likelihoodFieldSensorModel` for property details.

```
rangeFinderModel = likelihoodFieldSensorModel;
rangeFinderModel.SensorLimits = [0.45 8];
rangeFinderModel.Map = map;
```

Set `rangeFinderModel.SensorPose` to the coordinate transform of the fixed camera with respect to the robot base. This is used to transform the laser readings from camera frame to the base frame of TurtleBot. Please refer to "Access the tf Transformation Tree in ROS" (ROS Toolbox) for details on coordinate transformations.

Note that currently `SensorModel` is only compatible with sensors that are fixed on the robot's frame, which means the sensor transform is constant.

```
% Query the Transformation Tree (tf tree) in ROS.
tftree = rostf;
waitForTransform(tftree, '/base_link', '/base_scan');
sensorTransform = getTransform(tftree, '/base_link', '/base_scan');

% Get the euler rotation angles.
laserQuat = [sensorTransform.Transform.Rotation.W sensorTransform.Transform.Rotation.X ...
            sensorTransform.Transform.Rotation.Y sensorTransform.Transform.Rotation.Z];
laserRotation = quat2eul(laserQuat, 'ZYX');

% Setup the |SensorPose|, which includes the translation along base_link's
% +X, +Y direction in meters and rotation angle along base_link's +Z axis
% in radians.
rangeFinderModel.SensorPose = ...
    [sensorTransform.Transform.Translation.X sensorTransform.Transform.Translation.Y laserRotati
```

Receiving Sensor Measurements and Sending Velocity Commands

Create ROS subscribers for retrieving sensor and odometry measurements from TurtleBot.

```
laserSub = rossubscriber('/scan');
odomSub = rossubscriber('/odom');
```

Create ROS publisher for sending out velocity commands to TurtleBot. TurtleBot subscribes to `' / mobile_base/commands/velocity'` for velocity commands.

```
[velPub, velMsg] = ...
    rospublisher('/cmd_vel', 'geometry_msgs/Twist');
```

Initialize AMCL Object

Instantiate an AMCL object `amcl`. See `monteCarloLocalization` for more information on the class.

```
amcl = monteCarloLocalization;
amcl.UseLidarScan = true;
```

Assign the `MotionModel` and `SensorModel` properties in the `amcl` object.

```
amcl.MotionModel = odometryModel;
amcl.SensorModel = rangeFinderModel;
```

The particle filter only updates the particles when the robot's movement exceeds the `UpdateThresholds`, which defines minimum displacement in `[x, y, yaw]` to trigger filter update. This prevents too frequent updates due to sensor noise. Particle resampling happens after the `amcl.ResamplingInterval` filter updates. Using larger numbers leads to slower particle depletion at the price of slower particle convergence as well.

```
amcl.UpdateThresholds = [0.2, 0.2, 0.2];
amcl.ResamplingInterval = 1;
```

Configure AMCL Object for Localization with Initial Pose Estimate.

`amcl.ParticleLimits` defines the lower and upper bound on the number of particles that will be generated during the resampling process. Allowing more particles to be generated may improve the

chance of converging to the true robot pose, but has an impact on computation speed and particles may take longer time or even fail to converge. Please refer to the 'KL-D Sampling' section in [1] for computing a reasonable bound value on the number of particles. Note that global localization may need significantly more particles compared to localization with an initial pose estimate. If the robot knows its initial pose with some uncertainty, such additional information can help AMCL localize robots faster with a less number of particles, i.e. you can use a smaller value of upper bound in `amcl.ParticleLimits`.

Now set `amcl.GlobalLocalization` to false and provide an estimated initial pose to AMCL. By doing so, AMCL holds the initial belief that robot's true pose follows a Gaussian distribution with a mean equal to `amcl.InitialPose` and a covariance matrix equal to `amcl.InitialCovariance`. Initial pose estimate should be obtained according to your setup. This example helper retrieves the robot's current true pose from Gazebo.

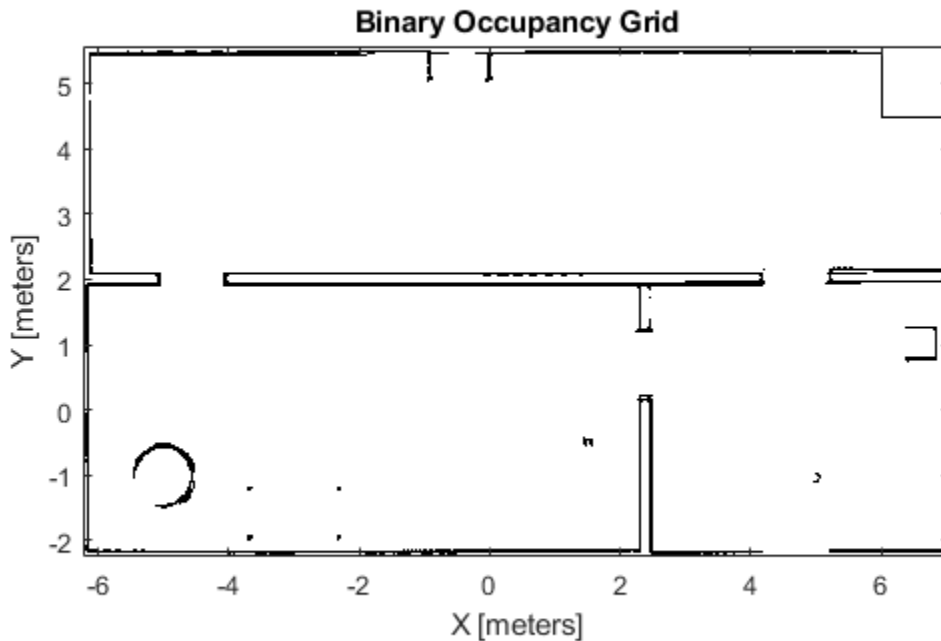
Please refer to section **Configure AMCL object for global localization** for an example on using global localization.

```
amcl.ParticleLimits = [500 5000];
amcl.GlobalLocalization = false;
amcl.InitialPose = ExampleHelperAMCLGazeboTruePose;
amcl.InitialCovariance = eye(3)*0.5;
```

Setup Helper for Visualization and Driving TurtleBot.

Setup `ExampleHelperAMCLVisualization` to plot the map and update robot's estimated pose, particles, and laser scan readings on the map.

```
visualizationHelper = ExampleHelperAMCLVisualization(map);
```



Robot motion is essential for the AMCL algorithm. In this example, we drive TurtleBot randomly using the `ExampleHelperAMCLWanderer` class, which drives the robot inside the environment while avoiding obstacles using the `controllerVFH` class.

```
wanderHelper = ...
  ExampleHelperAMCLWanderer(laserSub, sensorTransform, velPub, velMsg);
```

Localization Procedure

The AMCL algorithm is updated with odometry and sensor readings at each time step when the robot is moving around. Please allow a few seconds before particles are initialized and plotted in the figure. In this example we will run `numUpdates` AMCL updates. If the robot doesn't converge to the correct robot pose, consider using a larger `numUpdates`.

```
numUpdates = 60;
i = 0;
while i < numUpdates
  % Receive laser scan and odometry message.
  scanMsg = receive(laserSub);
  odompose = odomSub.LatestMessage;

  % Create lidarScan object to pass to the AMCL object.
  scan = lidarScan(scanMsg);

  % For sensors that are mounted upside down, you need to reverse the
  % order of scan angle readings using 'flip' function.
```

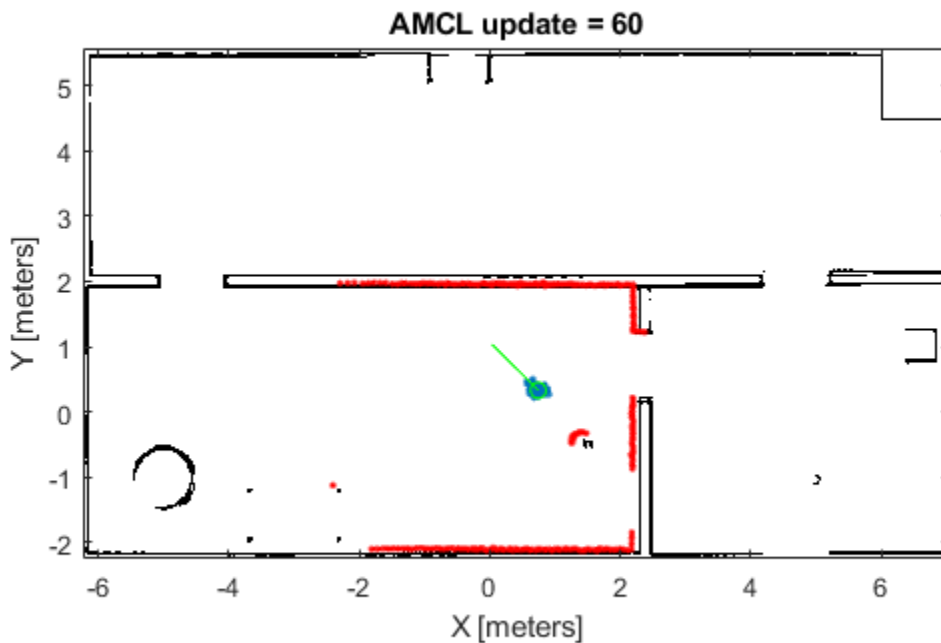
```

% Compute robot's pose [x,y,yaw] from odometry message.
odomQuat = [odompose.Pose.Pose.Orientation.W, odompose.Pose.Pose.Orientation.X, ...
            odompose.Pose.Pose.Orientation.Y, odompose.Pose.Pose.Orientation.Z];
odomRotation = quat2eul(odomQuat);
pose = [odompose.Pose.Pose.Position.X, odompose.Pose.Pose.Position.Y odomRotation(1)];

% Update estimated robot's pose and covariance using new odometry and
% sensor readings.
[isUpdated,estimatedPose, estimatedCovariance] = amcl(pose, scan);

% Drive robot to next pose.
wander(wanderHelper);

% Plot the robot's estimated pose, particles and laser scans on the map.
if isUpdated
    i = i + 1;
    plotStep(visualizationHelper, amcl, estimatedPose, scan, i)
end
end
end
    
```



Stop the TurtleBot and Shutdown ROS in MATLAB

```

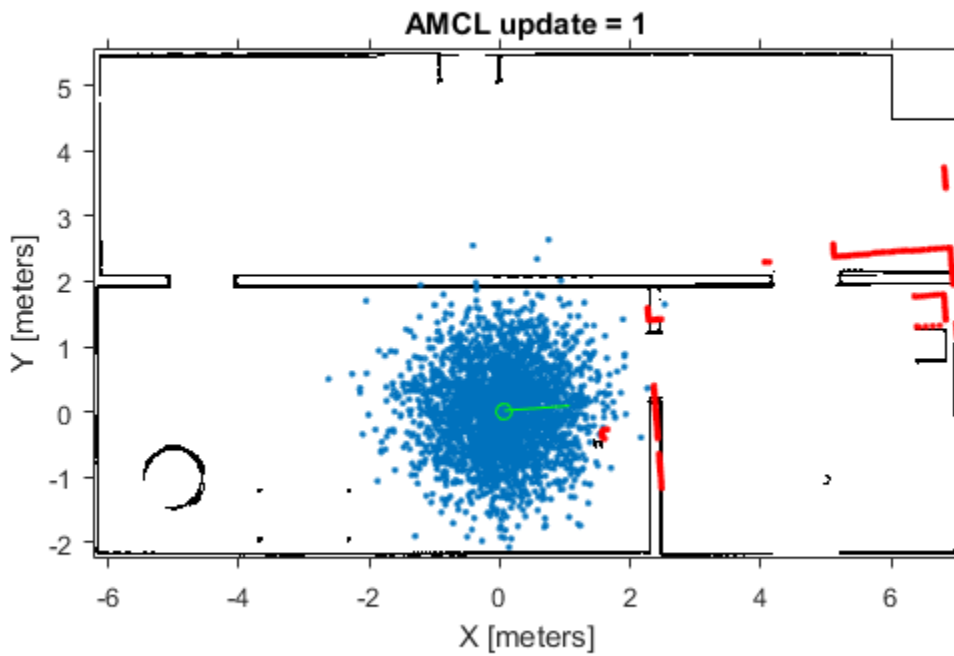
stop(wanderHelper);
rosshutdown
    
```

Shutting down global node /matlab_global_node_73841 with NodeURI <http://192.168.2.1:53461/>

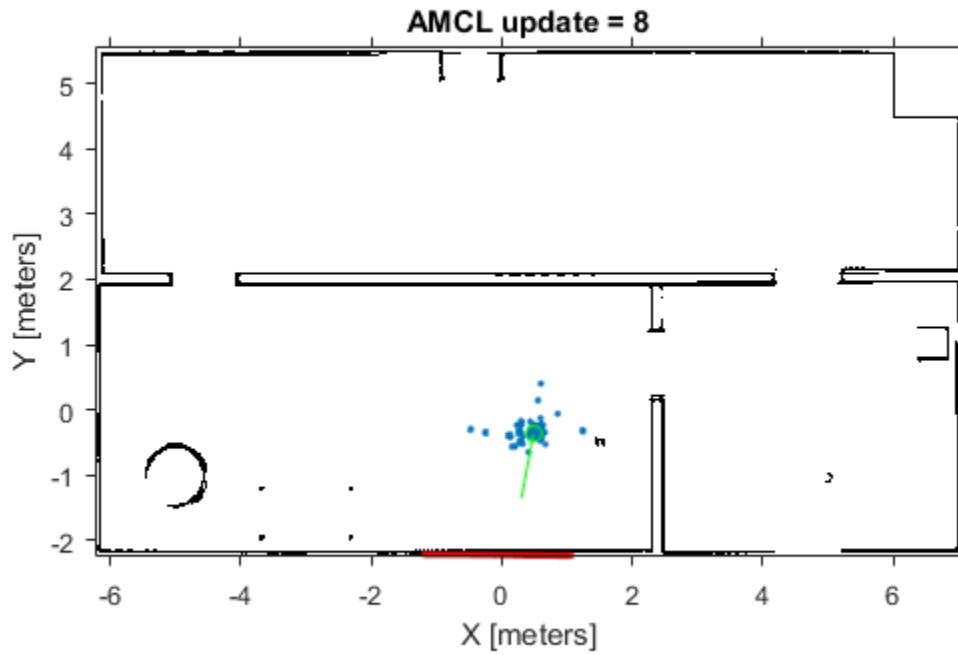
Sample Results for AMCL Localization with Initial Pose Estimate

AMCL is a probabilistic algorithm, the simulation result on your computer may be slightly different from the sample run shown here.

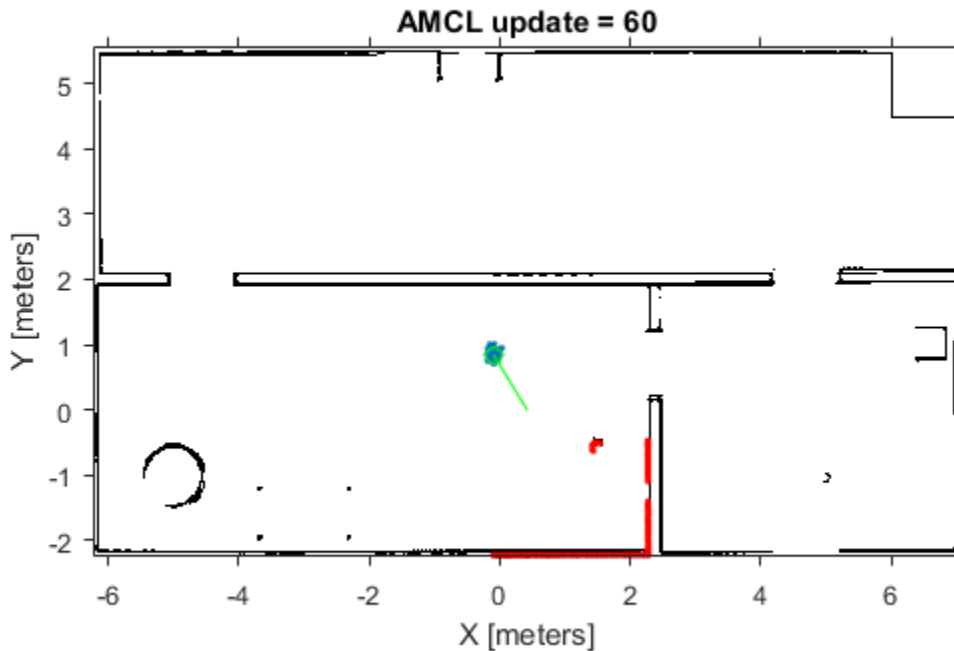
After first AMCL update, particles are generated by sampling Gaussian distribution with mean equal to `amcl.InitialPose` and covariance equal to `amcl.InitialCovariance`.



After 8 updates, the particles start converging to areas with higher likelihood:



After 60 updates, all particles should converge to the correct robot pose and the laser scans should closely align with the map outlines.



Configure AMCL Object for Global Localization.

In case no initial robot pose estimate is available, AMCL will try to localize robot without knowing the robot's initial position. The algorithm initially assumes that the robot has equal probability in being anywhere in the office's free space and generates uniformly distributed particles inside such space. Thus Global localization requires significantly more particles compared to localization with initial pose estimate.

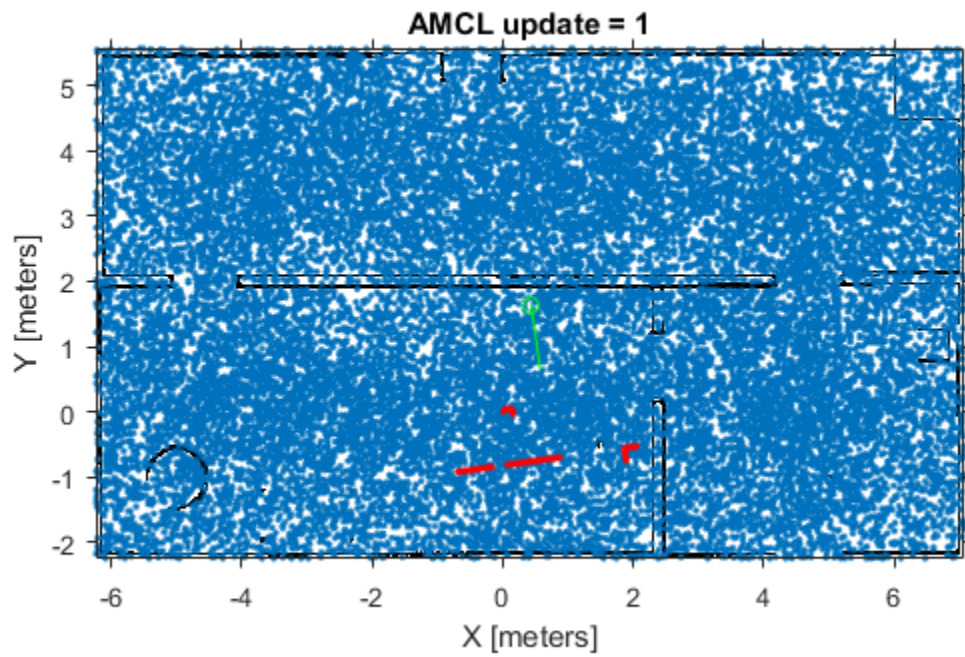
To enable AMCL global localization feature, replace the code sections in **Configure AMCL object for localization with initial pose estimate** with the code in this section.

```
amcl.GlobalLocalization = true;
amcl.ParticleLimits = [500 50000];
```

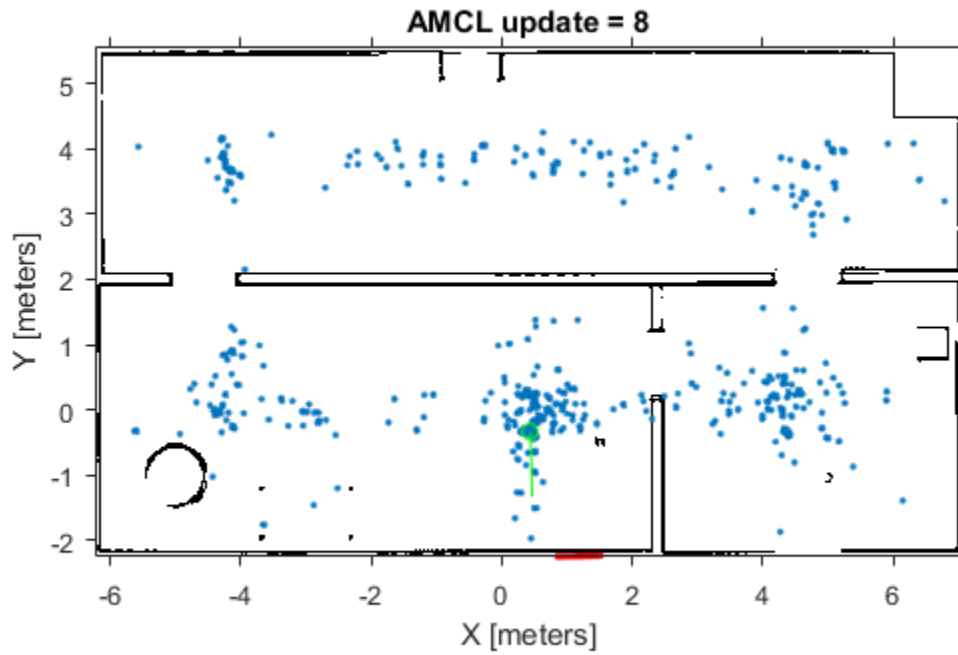
Sample Results for AMCL Global Localization

AMCL is a probabilistic algorithm, the simulation result on your computer may be slightly different from the sample run shown here.

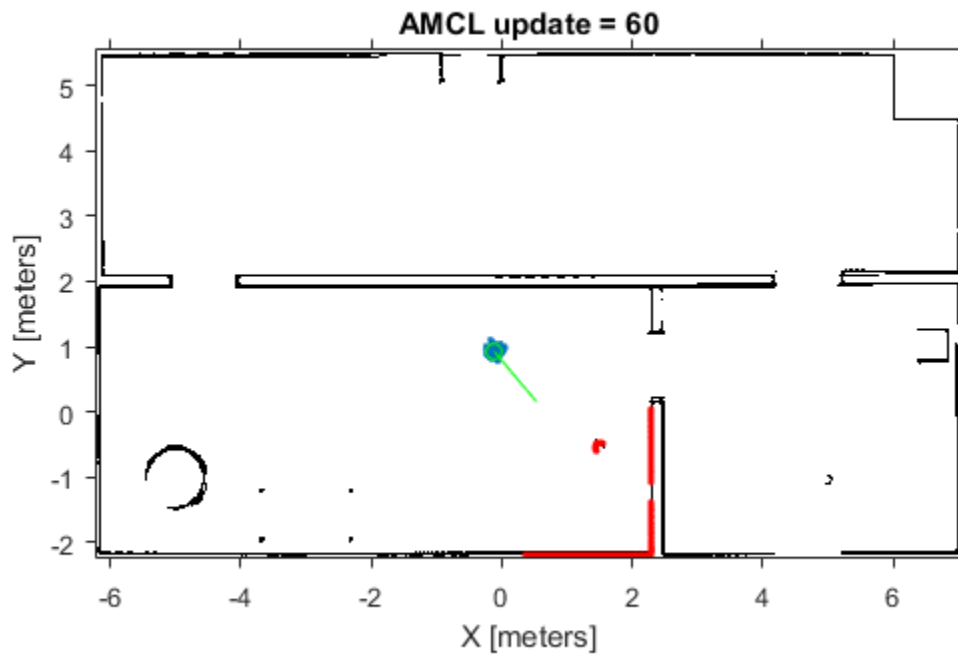
After first AMCL update, particles are uniformly distributed inside the free office space:



After 8 updates, the particles start converging to areas with higher likelihood:



After 60 updates, all particles should converge to the correct robot pose and the laser scans should closely align with the map outlines.



References

- [1] S. Thrun, W. Burgard and D. Fox, Probabilistic Robotics. Cambridge, MA: MIT Press, 2005.

Compose a Series of Laser Scans with Pose Changes

Use the `matchScans` function to compute the pose difference between a series of laser scans. Compose the relative poses by using a defined `composePoses` function to get a transformation to the initial frame. Then, transform all laser scans into the initial frame using these composed poses.

Specify the original laser scan and offsets to generate a series of shifted laser scans. Iterate through the scans and transform the original scan based on each offset. Plot the laser scans to see the shifted data.

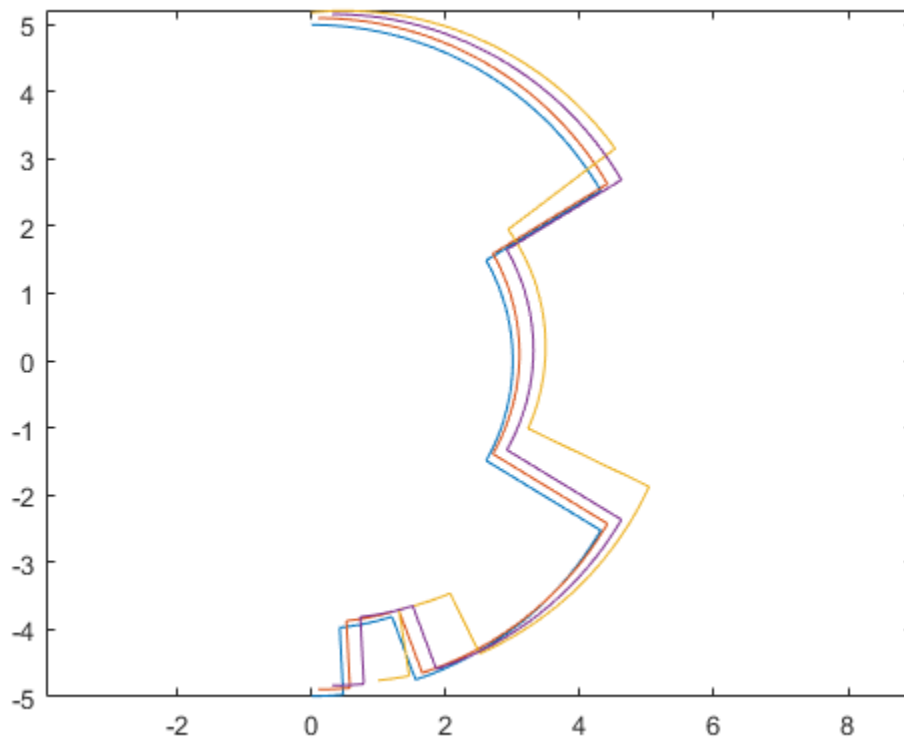
```

ranges = zeros(300,4);
angles = zeros(300,4);
ranges(:,1) = 5*ones(300,1);
ranges(11:30,1) = 4*ones(1,20);
ranges(101:200,1) = 3*ones(1,100);
angles(:,1) = linspace(-pi/2,pi/2,300);
offset(1,:) = [0.1 0.1 0];
offset(2,:) = [0.4 0.1 0.1];
offset(3,:) = [-0.2 0 -0.1];

for i = 2:4
    [ranges(:,i),angles(:,i)] = transformScan(ranges(:,i-1),angles(:,i-1),offset(i-1,:));
end

[x,y] = pol2cart(angles,ranges);
plot(x,y)
axis equal

```



Perform scan matching on each laser scan set to get the relative pose between each scan. The outputs from the `matchScans` function are close to the specified offsets. The initial scan is in the initial frame, so the pose difference is `[0 0 0]`.

```
relPoses(1,:) = [0 0 0];

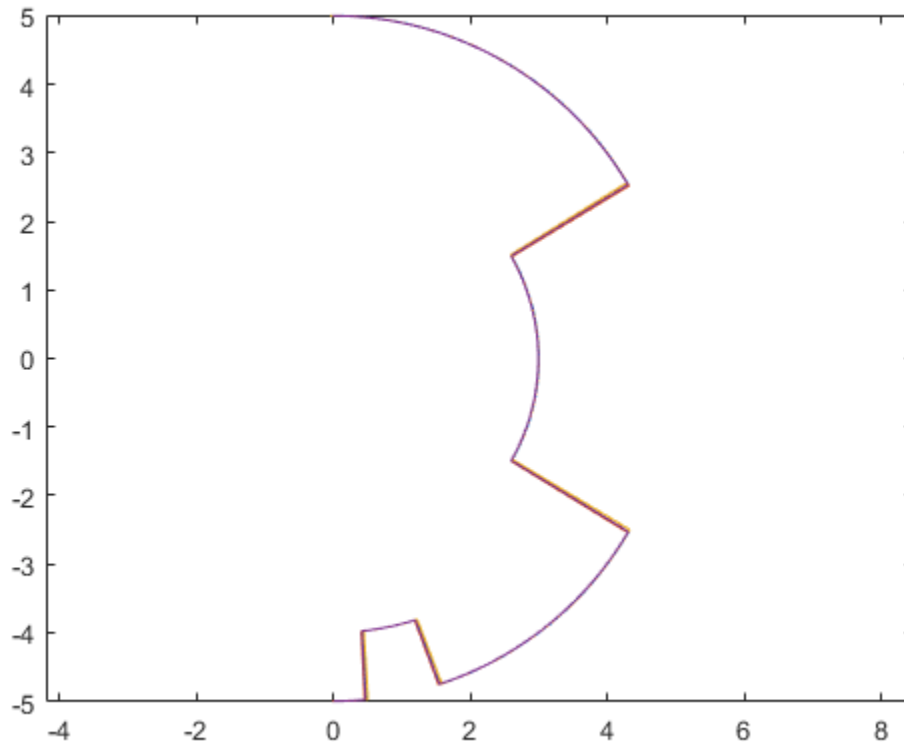
for i = 2:4
    relPoses(i,:) = matchScans(ranges(:,i),angles(:,i),...
                              ranges(:,i-1),angles(:,i-1),'CellSize',1);
end
```

Use the `composePoses` function in a loop to get the absolute transformation for each laser scan. This function is defined at the end of the example. Transform each scan to get them all in the initial frame.

```
transRanges = zeros(300,4);
transAngles = zeros(300,4);
transRanges(:,1) = ranges(:,1);
transAngles(:,1) = angles(:,1);
composedPoses(1,:) = [0 0 0];
for i = 2:4
    composedPoses(i,:) = composePoses(relPoses(i,:),composedPoses(i-1,:));
    [transRanges(:,i),transAngles(:,i)] = transformScan(ranges(:,i),angles(:,i),composedPoses(i,
end
```

Plot the transformed ranges and angles. They overlap well, based on the calculated transformations from `matchScans`.

```
[x,y] = pol2cart(transAngles,transRanges);
plot(x,y)
axis equal
```



Define the `composePoses` function. This function takes in the transformation of the initial frame to the base frame and the relative transformation from the initial frame to a second frame. For a series of laser scans, the relative input is the relative pose between the last two frames, and the base input is the composed pose over all previous scans.

You can also define this function in a separate script and save to the current folder.

```
function composedPose = composePoses(relative,base)
    %Convert both poses (3-by-1 vector) to transformations (4-by-4 matrix) and multiply
    %together using pose2tform function.
    tform = pose2tform(base)*pose2tform(relative);

    % Extract translational vector and Euler angles as ZYX.
    trvec = tform2trvec(tform);
    eul = tform2eul(tform);

    % Concatenate the elements of the transform as [x y theta].
    composedPose = [trvec(1:2) eul(1)];

    % Function to convert pose to transform.
    function tform = pose2tform(pose)
        x = pose(1);
        y = pose(2);
        th = wrapTo2Pi(pose(3));
        tform = trvec2tform([x y 0])*eul2tform([th 0 0]);
    end
end
```


See Also

`transformScan` | `matchScans`

Minimize Search Range in Grid-based Lidar Scan Matching Using IMU

This example shows how to use an inertial measurement unit (IMU) to minimize the search range of the rotation angle for scan matching algorithms. IMU sensor readings are used to estimate the orientation of the vehicle, and specified as the initial guess for the `matchScansGrid` function. This method of initial pose estimation is compared to the base algorithm with assumes an initial guess of `[0 0 0]`.

Load Logged Data

Load the MAT-file, `loggedLidarAndIMUData.mat`. This file contains lidar scans, accelerometer readings, and gyroscopes readings, and the corresponding timestamps.

```
rng(1); % Fixed RNG seed for repeatability
load('loggedLidarAndIMUData', ...
     'tLidar', 'lidarScans', ...
     'imuFs', 'tIMU', 'accel', 'gyro');

startIdx = 1;
endIdx = numel(lidarScans)-1;
```

Sync IMU Time Indices with Lidar Time Indices

The IMU and lidar update at different sampling rates. Create an array that maps lidar to IMU indices.

```
lidarToIMUIndices = zeros(size(tLidar));
for i = 1:numel(tLidar)
    [~, lidarToIMUIndices(i)] = min(abs(tLidar(i) - tIMU));
end
```

Estimate Yaw from IMU

Estimate the orientation from the accelerometer and gyroscope readings as a quaternion using the `imufilter` object. Then, calculate the relative yaws between successive lidar scans by converting the quaternions to Euler angles.

```
orientFilt = imufilter('SampleRate', imuFs);
q = orientFilt(accel, gyro);

% Calculate relative yaws
eulerAngs = euler(q(lidarToIMUIndices(1+(startIdx:endIdx))) ...
    .* conj(q(lidarToIMUIndices(startIdx:endIdx))), 'ZYX', 'frame');
imuYaws = eulerAngs(:,1);
```

Run Scan Matching and Log Results

Run the `matchScansGrid` function with two different options:

- Default initial guess and search range
- Initial guess based on IMU sensor readings with a small search range

Iterate through all the lidar scans readings, running `matchScansGrid` with each pair of sequential scans. Log the processing times for each function call and the relative pose outputs from scan matching. To visualize the transformed scans based on the solution, set `plotSolutions` to 1. However, in this example, the difference in pose between the two different options is not noticeable.

```

smallSearchRange = pi/8;
plotSolutions = 0;

% Initialize time values and relative pose arrays
timeDefaultSearch = NaN(endIdx - startIdx + 1,1);
timeSmallSearchWithIMU = NaN(endIdx - startIdx + 1,1);
allRelPosesDefault = NaN(endIdx - startIdx + 1,3);
allRelPosesIMU = NaN(endIdx - startIdx + 1,3);

for idx = startIdx:endIdx
    scan1 = lidarScans(idx);
    scan2 = lidarScans(idx+1);

    yaw = imuYaws(idx);
    initGuess = [0 0 yaw];

    % Run scan matching with default values.
    tic;
    relPose = matchScansGrid(scan2, scan1);
    timeDefaultSearch(idx) = toc;

    allRelPosesDefault(idx,:) = relPose;

    % Run scan matching with IMU-based initial yaw and small search range.
    tic;
    relPose = matchScansGrid(scan2, scan1, 'InitialPose', initGuess, ...
        'RotationSearchRange', smallSearchRange);
    timeSmallSearchWithIMU(idx) = toc;
    allRelPosesIMU(idx,:) = relPose;

    % Set plot solutions to 1 to turn on scan visualization.
    if plotSolutions == 1
        figure(cfg,'Visible','on')
        plot(scan1)
        hold on
        plot(transformScan(scan2, allRelPosesDefault(idx,:)))
        plot(transformScan(scan2, allRelPosesIMU(idx,:)))
        hold off
        legend('Ref Scan','Default', ...
            'Small Search Range + IMU',...
            'Location','northwest')
        title(sprintf('Matched Lidar Scans %d and %d', i, i+1))
    end
end
end

```

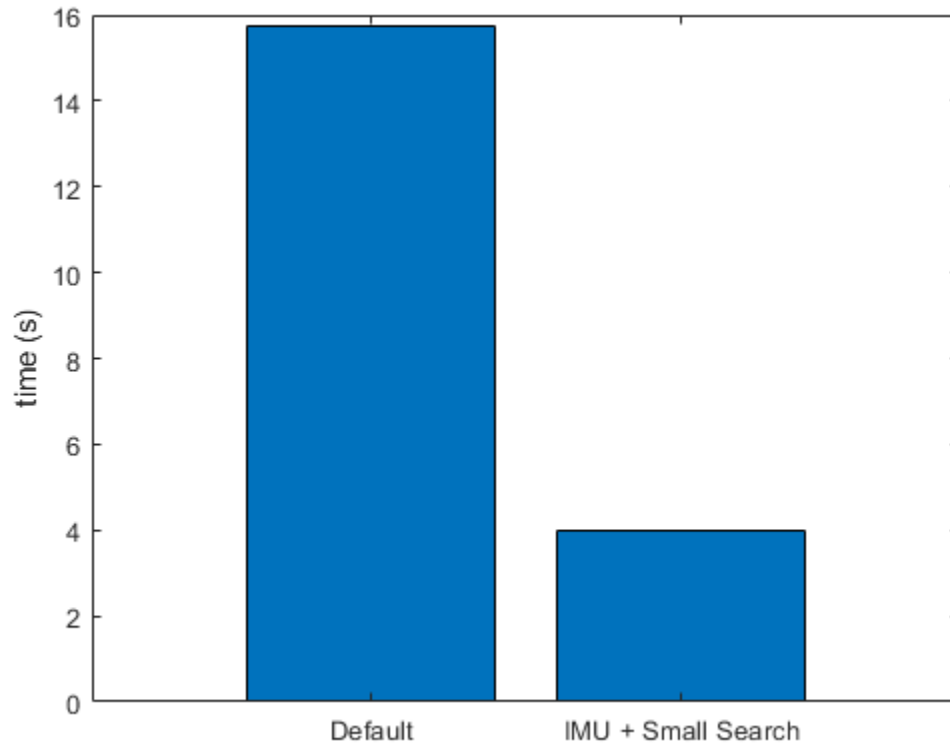
Compare Results

Visualize and compare the scan matching results. Show the total processing time as a bar chart. Then, compare each iteration's time.

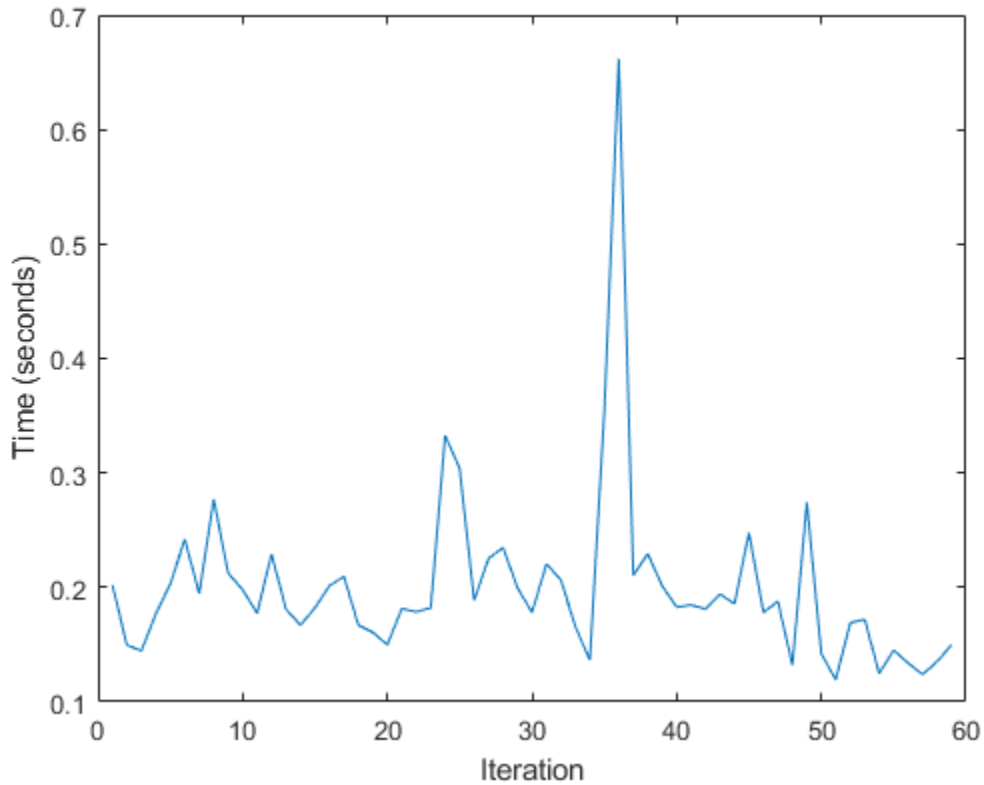
```

figure
title('Scan Matching Processing Time')
bar(categorical({'Default','IMU + Small Search'}), ...
    [sum(timeDefaultSearch),sum(timeSmallSearchWithIMU)])
ylabel('time (s)')

```

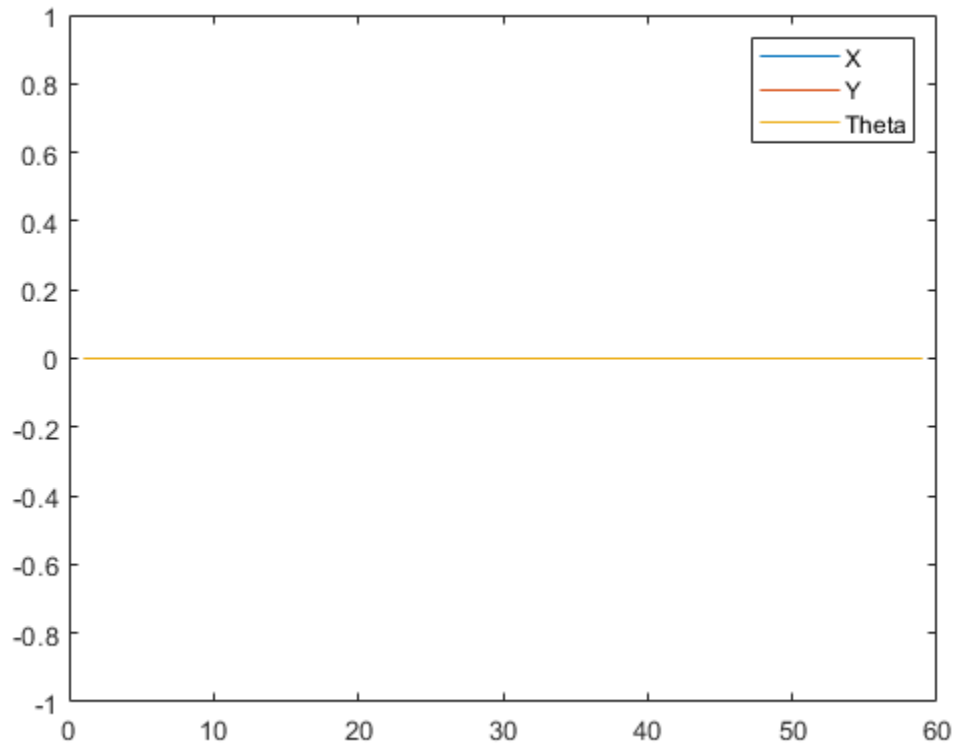


```
figure
title('Difference in Iteration Time')
plot(startIdx:endIdx,(timeDefaultSearch - timeSmallSearchWithIMU))
ylabel('Time (seconds)')
xlabel('Iteration')
```



Based on the timing results, specifying IMU sensor readings as an estimate to the scan matching algorithm improves the time of each iteration. As a final step, you can verify the difference in estimate pose is not significant. For this example, all poses from `matchScansGrid` are the same.

```
figure
title('Difference in Pose Values')
plot(allRelPosesDefault-allRelPosesIMU)
legend('X', 'Y', 'Theta')
```



Visual-Inertial Odometry Using Synthetic Data

This example shows how to estimate the pose (position and orientation) of a ground vehicle using an inertial measurement unit (IMU) and a monocular camera. In this example, you:

- 1 Create a driving scenario containing the ground truth trajectory of the vehicle.
- 2 Use an IMU and visual odometry model to generate measurements.
- 3 Fuse these measurements to estimate the pose of the vehicle and then display the results.

Visual-inertial odometry estimates pose by fusing the visual odometry pose estimate from the monocular camera and the pose estimate from the IMU. The IMU returns an accurate pose estimate for small time intervals, but suffers from large drift due to integrating the inertial sensor measurements. The monocular camera returns an accurate pose estimate over a larger time interval, but suffers from a scale ambiguity. Given these complementary strengths and weaknesses, the fusion of these sensors using visual-inertial odometry is a suitable choice. This method can be used in scenarios where GPS readings are unavailable, such as in an urban canyon.

Create a Driving Scenario with Trajectory

Create a `drivingScenario` (Automated Driving Toolbox) object that contains:

- The road the vehicle travels on
- The buildings surrounding either side of the road
- The ground truth pose of the vehicle
- The estimated pose of the vehicle

The ground truth pose of the vehicle is shown as a solid blue cuboid. The estimated pose is shown as a transparent blue cuboid. Note that the estimated pose does not appear in the initial visualization because the ground truth and estimated poses overlap.

Generate the baseline trajectory for the ground vehicle using the `waypointTrajectory` (Sensor Fusion and Tracking Toolbox) System object™. Note that the `waypointTrajectory` is used in place of `drivingScenario/trajectory` since the acceleration of the vehicle is needed. The trajectory is generated at a specified sampling rate using a set of waypoints, times of arrival, and velocities.

```
% Create the driving scenario with both the ground truth and estimated
% vehicle poses.
```

```
scene = drivingScenario;
groundTruthVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);
estVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);
```

```
% Generate the baseline trajectory.
```

```
sampleRate = 100;
wayPoints = [ 0 0 0;
             200 0 0;
             200 50 0;
             200 230 0;
             215 245 0;
             260 245 0;
             290 240 0;
             310 258 0;
             290 275 0;
             260 260 0;
             -20 260 0];
```

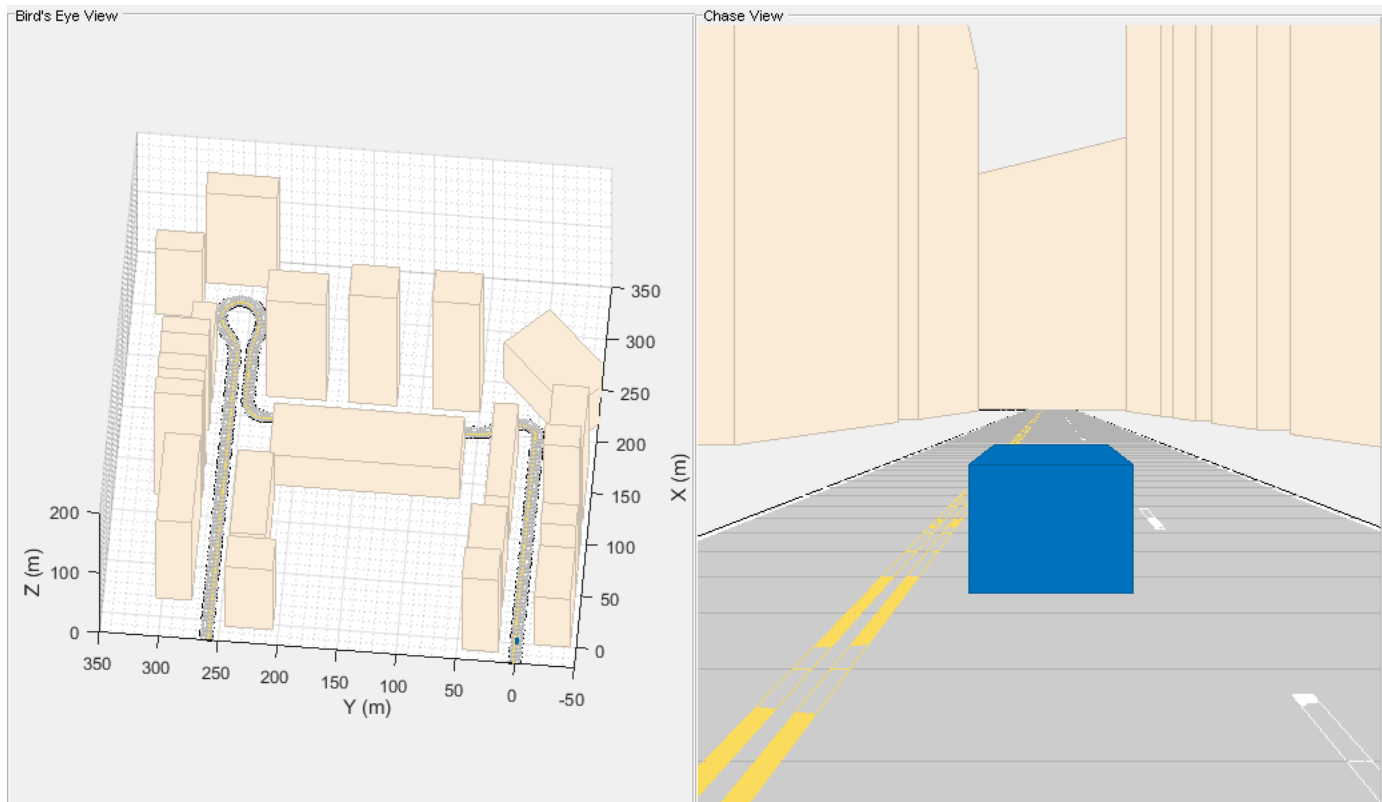
```

t = [0 20 25 44 46 50 54 56 59 63 90].';
speed = 10;
velocities = [ speed    0 0;
              speed    0 0;
              0 speed 0;
              0 speed 0;
              speed    0 0;
              speed    0 0;
              speed    0 0;
              0 speed 0;
              -speed   0 0;
              -speed   0 0;
              -speed   0 0];

traj = waypointTrajectory(wayPoints, 'TimeOfArrival', t, ...
    'Velocities', velocities, 'SampleRate', sampleRate);

% Add a road and buildings to scene and visualize.
helperPopulateScene(scene, groundTruthVehicle);

```



Create a Fusion Filter

Create the filter to fuse IMU and visual odometry measurements. This example uses a loosely coupled method to fuse the measurements. While the results are not as accurate as a tightly coupled method, the amount of processing required is significantly less and the results are adequate. The fusion filter uses an error-state Kalman filter to track orientation (as a quaternion), position, velocity, and sensor biases.

The `insfilterErrorState` object has the following functions to process sensor data: `predict` and `fusemvo`.

The `predict` function takes the accelerometer and gyroscope measurements from the IMU as inputs. Call the `predict` function each time the accelerometer and gyroscope are sampled. This function predicts the state forward by one time step based on the accelerometer and gyroscope measurements, and updates the error state covariance of the filter.

The `fusemvo` function takes the visual odometry pose estimates as input. This function updates the error states based on the visual odometry pose estimates by computing a Kalman gain that weighs the various inputs according to their uncertainty. As with the `predict` function, this function also updates the error state covariance, this time taking the Kalman gain into account. The state is then updated using the new error state and the error state is reset.

```
filt = insfilterErrorState('IMUSampleRate', sampleRate, ...
    'ReferenceFrame', 'ENU')
% Set the initial state and error state covariance.
helperInitialize(filt, traj);
```

```
filt =
```

```
insfilterErrorState with properties:
```

```
    IMUSampleRate: 100          Hz
ReferenceLocation: [0 0 0]     [deg deg m]
           State: [17x1 double]
           StateCovariance: [16x16 double]

Process Noise Variances
           GyroscopeNoise: [1e-06 1e-06 1e-06]    (rad/s)2
           AccelerometerNoise: [0.0001 0.0001 0.0001] (m/s2)2
           GyroscopeBiasNoise: [1e-09 1e-09 1e-09] (rad/s)2
           AccelerometerBiasNoise: [0.0001 0.0001 0.0001] (m/s2)2
```

Specify the Visual Odometry Model

Define the visual odometry model parameters. These parameters model a feature matching and tracking-based visual odometry system using a monocular camera. The `scale` parameter accounts for the unknown scale of subsequent vision frames of the monocular camera. The other parameters model the drift in the visual odometry reading as a combination of white noise and a first-order Gauss-Markov process.

```
% The flag useVO determines if visual odometry is used:
% useVO = false; % Only IMU is used.
useVO = true; % Both IMU and visual odometry are used.

paramsVO.scale = 2;
paramsVO.sigmaN = 0.139;
paramsVO.tau = 232;
paramsVO.sigmaB = sqrt(1.34);
paramsVO.driftBias = [0 0 0];
```

Specify the IMU Sensor

Define an IMU sensor model containing an accelerometer and gyroscope using the `imuSensor` System object. The sensor model contains properties to model both deterministic and stochastic noise sources. The property values set here are typical for low-cost MEMS sensors.

```
% Set the RNG seed to default to obtain the same results for subsequent
% runs.
rng('default')

imu = imuSensor('SampleRate', sampleRate, 'ReferenceFrame', 'ENU');

% Accelerometer
imu.Accelerometer.MeasurementRange = 19.6; % m/s^2
imu.Accelerometer.Resolution = 0.0024; % m/s^2/LSB
imu.Accelerometer.NoiseDensity = 0.01; % (m/s^2)/sqrt(Hz)

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250); % rad/s
imu.Gyroscope.Resolution = deg2rad(0.0625); % rad/s/LSB
imu.Gyroscope.NoiseDensity = deg2rad(0.0573); % (rad/s)/sqrt(Hz)
imu.Gyroscope.ConstantBias = deg2rad(2); % rad/s
```

Set Up the Simulation

Specify the amount of time to run the simulation and initialize variables that are logged during the simulation loop.

```
% Run the simulation for 60 seconds.
numSecondsToSimulate = 60;
numIMUSamples = numSecondsToSimulate * sampleRate;

% Define the visual odometry sampling rate.
imuSamplesPerCamera = 4;
numCameraSamples = ceil(numIMUSamples / imuSamplesPerCamera);

% Preallocate data arrays for plotting results.
[pos, orient, vel, acc, angvel, ...
 posV0, orientV0, ...
 posEst, orientEst, velEst] ...
= helperPreallocateData(numIMUSamples, numCameraSamples);

% Set measurement noise parameters for the visual odometry fusion.
RposV0 = 0.1;
RorientV0 = 0.1;
```

Run the Simulation Loop

Run the simulation at the IMU sampling rate. Each IMU sample is used to predict the filter's state forward by one time step. Once a new visual odometry reading is available, it is used to correct the current filter state.

There is some drift in the filter estimates that can be further corrected with an additional sensor such as a GPS or an additional constraint such as a road boundary map.

```
cameraIdx = 1;
for i = 1:numIMUSamples
    % Generate ground truth trajectory values.
```

```

[pos(i,:), orient(i,:), vel(i,:), acc(i,:), angvel(i,:)] = traj();

% Generate accelerometer and gyroscope measurements from the ground truth
% trajectory values.
[accelMeas, gyroMeas] = imu(acc(i,:), angvel(i,:), orient(i));

% Predict the filter state forward one time step based on the
% accelerometer and gyroscope measurements.
predict(filt, accelMeas, gyroMeas);

if (1 == mod(i, imuSamplesPerCamera)) && useVO
    % Generate a visual odometry pose estimate from the ground truth
    % values and the visual odometry model.
    [posVO(cameraIdx,:), orientVO(cameraIdx,:), paramsVO] = ...
        helperVisualOdometryModel(pos(i,:), orient(i,:), paramsVO);

    % Correct filter state based on visual odometry data.
    fusemvo(filt, posVO(cameraIdx,:), RposVO, ...
        orientVO(cameraIdx), RorientVO);

    cameraIdx = cameraIdx + 1;
end

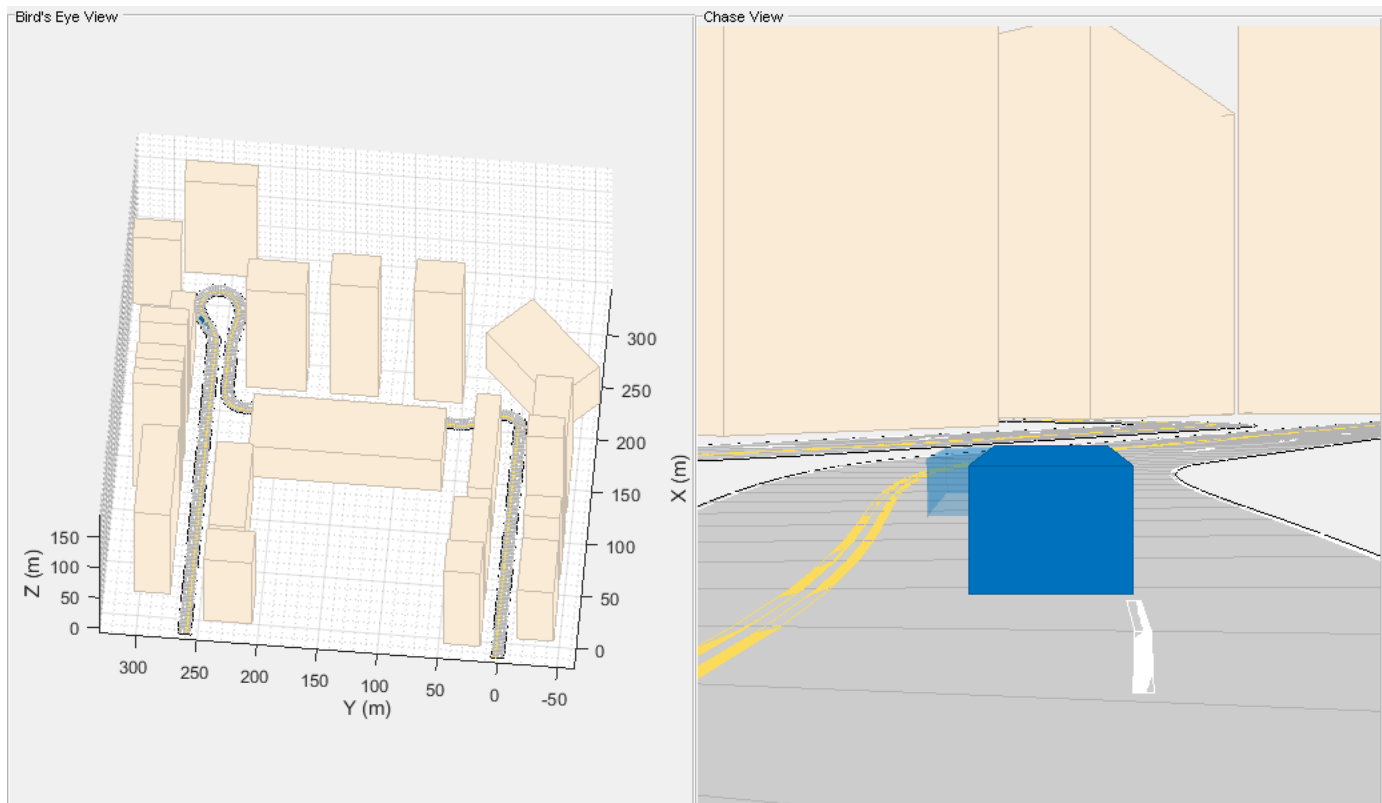
[posEst(i,:), orientEst(i,:), velEst(i,:)] = pose(filt);

% Update estimated vehicle pose.
helperUpdatePose(estVehicle, posEst(i,:), velEst(i,:), orientEst(i));

% Update ground truth vehicle pose.
helperUpdatePose(groundTruthVehicle, pos(i,:), vel(i,:), orient(i));

% Update driving scenario visualization.
updatePlots(scene);
drawnow limitrate;
end

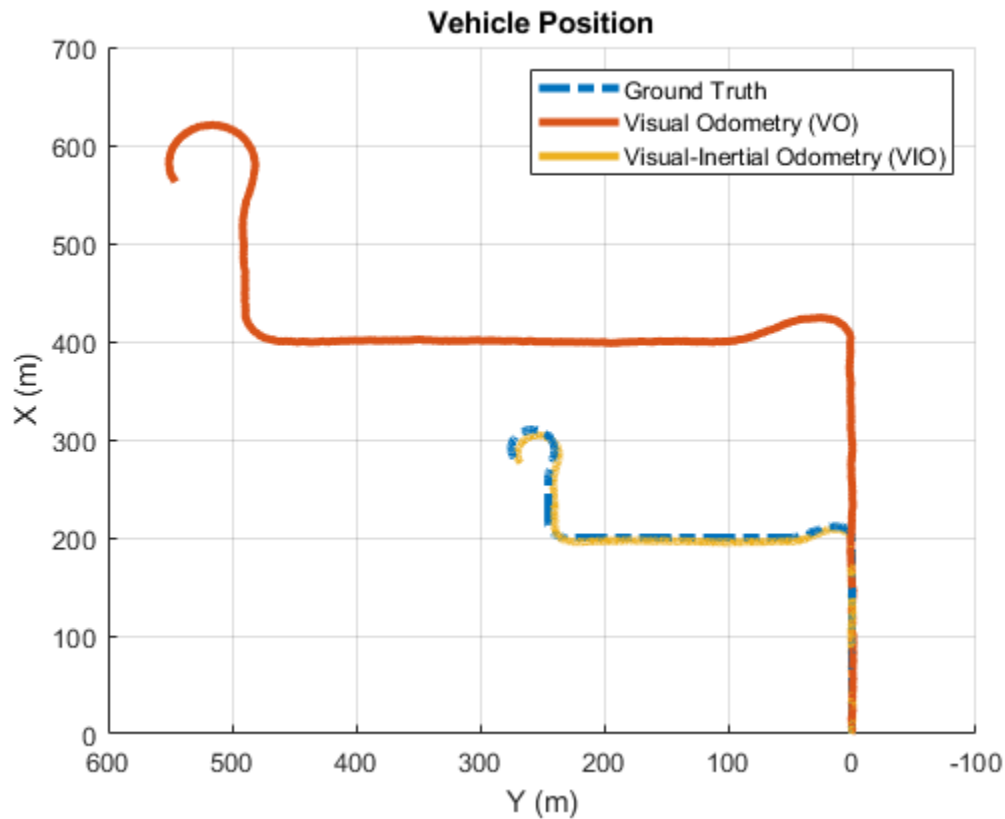
```



Plot the Results

Plot the ground truth vehicle trajectory, the visual odometry estimate, and the fusion filter estimate.

```
figure
if useV0
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
          posV0(:,1), posV0(:,2), posV0(:,3), ...
          posEst(:,1), posEst(:,2), posEst(:,3), ...
          'LineWidth', 3)
    legend('Ground Truth', 'Visual Odometry (V0)', ...
          'Visual-Inertial Odometry (VI0)', 'Location', 'northeast')
else
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
          posEst(:,1), posEst(:,2), posEst(:,3), ...
          'LineWidth', 3)
    legend('Ground Truth', 'IMU Pose Estimate')
end
view(-90, 90)
title('Vehicle Position')
xlabel('X (m)')
ylabel('Y (m)')
grid on
```



The plot shows that the visual odometry estimate is relatively accurate in estimating the shape of the trajectory. The fusion of the IMU and visual odometry measurements removes the scale factor uncertainty from the visual odometry measurements and the drift from the IMU measurements.

Supporting Functions

helperVisualOdometryModel

Compute visual odometry measurement from ground truth input and parameters struct. To model the uncertainty in the scaling between subsequent frames of the monocular camera, a constant scaling factor combined with a random drift is applied to the ground truth position.

```
function [posV0, orientV0, paramsV0] ...
    = helperVisualOdometryModel(pos, orient, paramsV0)
```

```
% Extract model parameters.
scaleV0 = paramsV0.scale;
sigmaN = paramsV0.sigmaN;
tau = paramsV0.tau;
sigmaB = paramsV0.sigmaB;
sigmaA = sqrt((2/tau) + 1/(tau*tau))*sigmaB;
b = paramsV0.driftBias;
```

```
% Calculate drift.
b = (1 - 1/tau).*b + randn(1,3)*sigmaA;
drift = randn(1,3)*sigmaN + b;
paramsV0.driftBias = b;
```

```
% Calculate visual odometry measurements.
posV0 = scaleV0*pos + drift;
orientV0 = orient;
end
```

helperInitialize

Set the initial state and covariance values for the fusion filter.

```
function helperInitialize(filt, traj)

% Retrieve the initial position, orientation, and velocity from the
% trajectory object and reset the internal states.
[pos, orient, vel] = traj();
reset(traj);

% Set the initial state values.
filt.State(1:4) = compact(orient(1)).';
filt.State(5:7) = pos(1,:).';
filt.State(8:10) = vel(1,:).';

% Set the gyroscope bias and visual odometry scale factor covariance to
% large values corresponding to low confidence.
filt.StateCovariance(10:12,10:12) = 1e6;
filt.StateCovariance(end) = 2e2;
end
```

helperPreallocateData

Preallocate data to log simulation results.

```
function [pos, orient, vel, acc, angvel, ...
         posV0, orientV0, ...
         posEst, orientEst, velEst] ...
         = helperPreallocateData(numIMUSamples, numCameraSamples)

% Specify ground truth.
pos = zeros(numIMUSamples, 3);
orient = quaternion.zeros(numIMUSamples, 1);
vel = zeros(numIMUSamples, 3);
acc = zeros(numIMUSamples, 3);
angvel = zeros(numIMUSamples, 3);

% Visual odometry output.
posV0 = zeros(numCameraSamples, 3);
orientV0 = quaternion.zeros(numCameraSamples, 1);

% Filter output.
posEst = zeros(numIMUSamples, 3);
orientEst = quaternion.zeros(numIMUSamples, 1);
velEst = zeros(numIMUSamples, 3);
end
```

helperUpdatePose

Update the pose of the vehicle.

```
function helperUpdatePose(veh, pos, vel, orient)

veh.Position = pos;
veh.Velocity = vel;
rpy = eulerd(orient, 'ZYX', 'frame');
veh.Yaw = rpy(1);
veh.Pitch = rpy(2);
veh.Roll = rpy(3);
end
```

References

- Sola, J. "Quaternion Kinematics for the Error-State Kalman Filter." ArXiv e-prints, arXiv:1711.02508v1 [cs.RO] 3 Nov 2017.
- R. Jiang, R., R. Klette, and S. Wang. "Modeling of Unbounded Long-Range Drift in Visual Odometry." 2010 Fourth Pacific-Rim Symposium on Image and Video Technology. Nov. 2010, pp. 121-126.

Landmark SLAM Using AprilTag Markers

This example shows how to combine robot odometry data and observed fiducial markers called AprilTags to better estimate the robot trajectory and the landmark positions in the environment.

Download Dataset

Download the `.mat` file that contains the raw data recorded from a rosbag on a ClearPath Robotics™ Jackal™. The raw data includes:

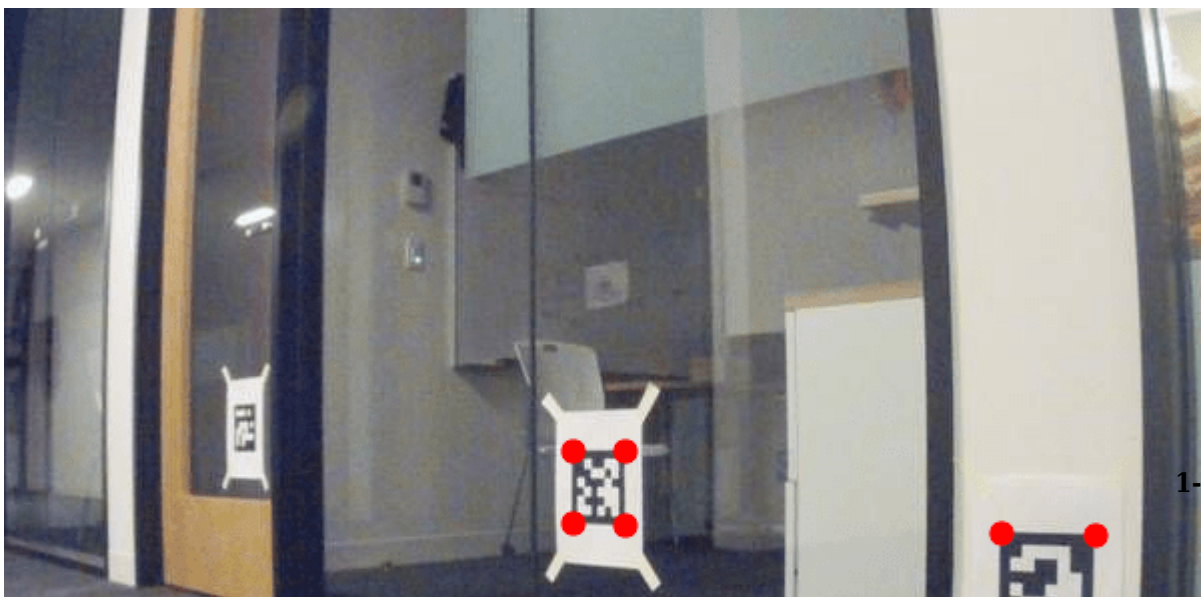
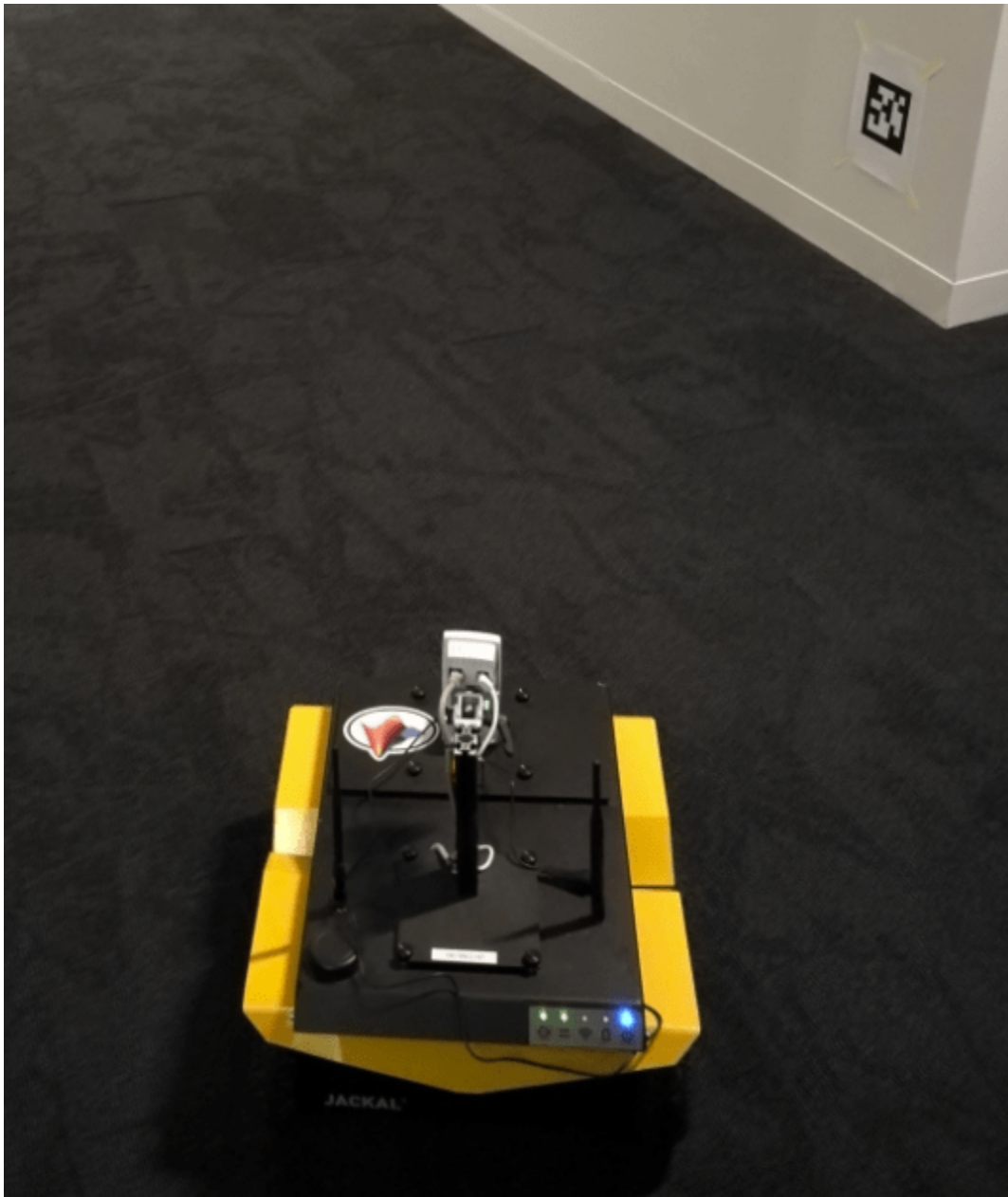
- Images taken using an Axis™ M1013 network camera at the resolution of 640x480.
- Odometry data pre-processed and synchronized with the image data.

The rosbag is imported into MATLAB™ and stored as a `.mat` file. To see how the data is extracted from the rosbag and pre-processed, the `exampleHelperExtractDataFromRosbag` example helper is provided at the end of this example. ROS Toolbox is required to use the helper function.

```
filename = matlab.internal.examples.downloadSupportFile("spc/robotics","apriltag_odometry_slam_d...  
unzip(filename);
```

Build Pose Graph From Sensor Data

In this example, a set of AprilTag markers are printed and randomly placed in the test environment. The tags are treated as *landmarks* in the pose graph. Given the camera intrinsics and the size of the tag, you can convert the images with AprilTag observations into point landmark measurements using the `readAprilTag` (Computer Vision Toolbox) function. These landmark measurements are relative to the current robot frame. To find out the intrinsic parameters of your camera, follow the steps in the “Camera Calibration Using AprilTag Markers” (Computer Vision Toolbox) example or use a checkerboard pattern with the Camera Calibrator (Computer Vision Toolbox) app. The AprilTag markers used in this example are taken from the “tag46h11” family with no duplicate IDs. The tag size printed is 136.65 mm.



```
tagFamily = "tag36h11";
tagSize = 136.65; % mm
```

```
load apriltag_odometry_slam_dataset/apriltag_odometry_slam_dataset.mat
load cameraParams.mat
```

After importing the synchronized sensor data and the camera parameters, build a pose graph of node estimates from the measurements in the sensor data. The pose graph contains node estimates, edge constraints, and loop closures that estimate robot poses.

```
% Create a pose graph object
pg = poseGraph3D;
```

By using fiducial markers like AprilTags, the block pattern in the image provides unique IDs for each landmark observation. This ID information reduces the need for difficult data association algorithms when performing simultaneous localization and mapping (SLAM).

Create the poseGraph3D object. Create variables to store the previous pose and node ID.

```
lastTform = [];
lastPoseNodeId = 1;
```

Create a containers.Map data structure to maintain the mapping between tag IDs and their node ID in the pose graph.

```
tagToNodeIDMap = containers.Map('KeyType','double','ValueType','double');
```

```
% Offset transformations for AprilTag observations
% The odometry data recorded in the .mat file is measured from 'odom'
% frame (i.e. fixed world frame) to the 'laser' frame that moves with
% the robot. There is a fixed transformation between the 'laser'
% frame and the camera frame applied manually.
R1 = [0 0 1; -1 0 0; 0 -1 0];
Ta = blkdiag(R1,1); % The camera frame has z axis pointing forward and y axis pointing down
Tb = eye(4); T2(1,3) = -0.11; T(3,3) = 0.146; % Fixed translation of camera frame origin to 'laser'
```

This example estimates the robot trajectory based on the landmark measurements from both the odometry and the AprilTag observations. Iterate through the sensor data and follow these steps:

- Add odometry data to pose graph using the `addRelativePose` function. Compute the relative poses between each odometry before adding to the pose graph.
- Add landmark measurements from the AprilTag observations in the images using the `readAprilTag` function. Because the image may contain multiple tags, iterate through all the IDs returned. Add point landmarks relative to the current pose node using the `addPointLandMark` function.
- Show the image with markers around the AprilTag observations. The image updates as you iterate through the data.

```
figure('Visible','on')
for i = 1:numel(imageData)

    % Add odometry data to pose graph
    T = odomData{i};
    if isempty(lastTform)
        nodePair = addRelativePose(pg,[0 0 0 1 0 0 0],[],lastPoseNodeId);
    else
```

```

        relPose = exampleHelperComputeRelativePose(lastTform,T);
        nodePair = addRelativePose(pg,relPose,[],lastPoseNodeId);
    end
    currPoseNodeId = nodePair(2);

    % Add landmark measurement based on AprilTag observation in the image.
    I = imageData{i};
    [id,loc,poseRigid3d,detectedFamily] = readAprilTag(I,tagFamily,cameraParams.Intrinsics,tagSi

    for j = 1:numel(id)
        % Insert observation markers to image.
        markerRadius = 6;
        numCorners = size(loc,1);
        markerPosition = [loc(:, :, j), repmat(markerRadius, numCorners, 1)];
        I = insertShape(I, "FilledCircle", markerPosition, "Color", "red", "Opacity", 1);

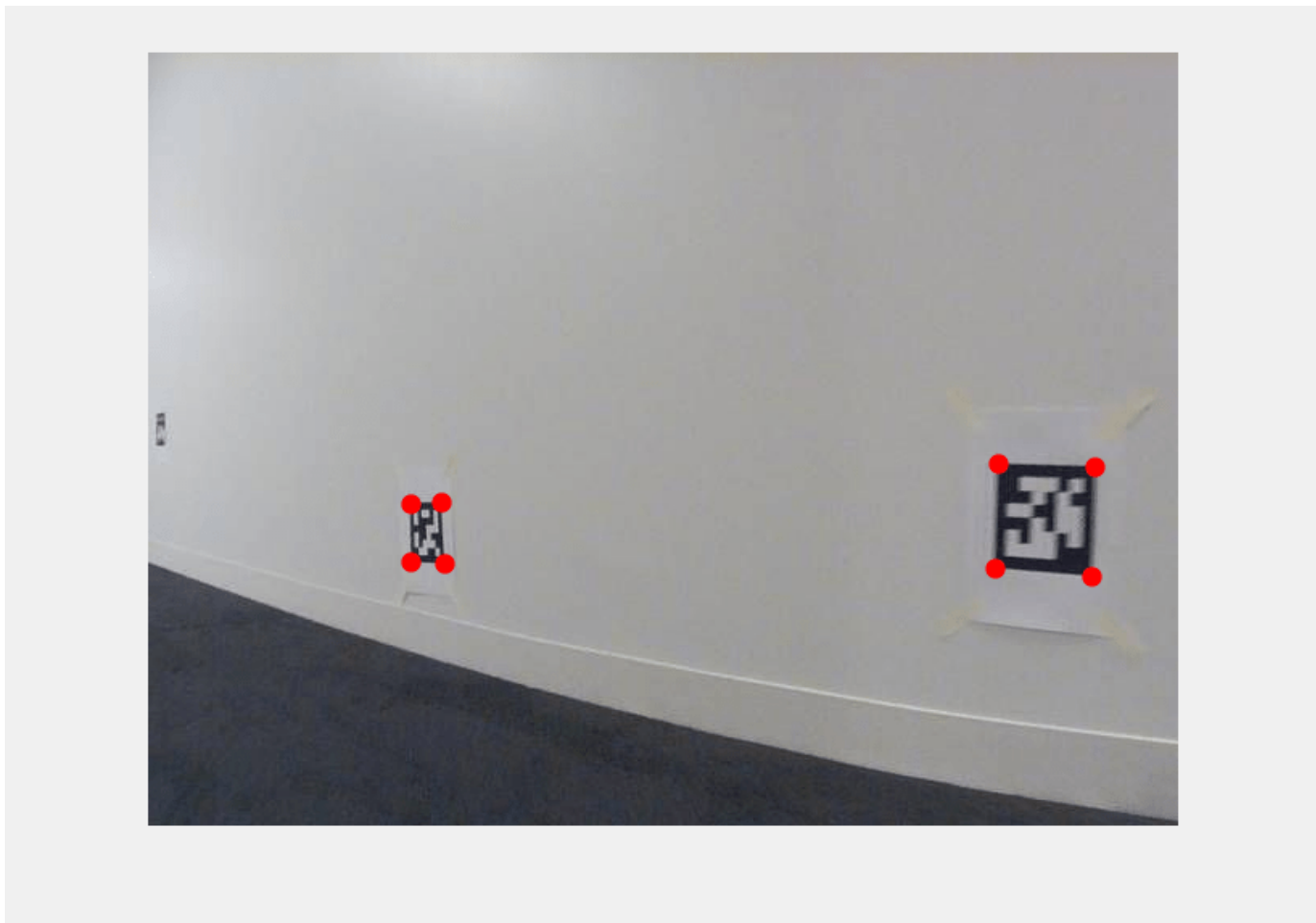
        t = poseRigid3d(j).Translation/1000; % change from mm to meter
        po = [t(:);1];
        p = Tb*Ta*po;

        if tagToNodeIDMap.isKey(id(j))
            lmkNodeId = tagToNodeIDMap(id(j));
            addPointLandmark(pg, p(1:3)', [], currPoseNodeId, lmkNodeId);
        else
            nodePair = addPointLandmark(pg, p(1:3)', [], currPoseNodeId);
            tagToNodeIDMap(id(j)) = nodePair(2);
        end
    end
end

% Show the image with AprilTag observation markers.
imshow(I)
drawnow

lastTform = T;
lastPoseNodeId = currPoseNodeId;
end

```



Optimize Pose Graph and Inspect Results

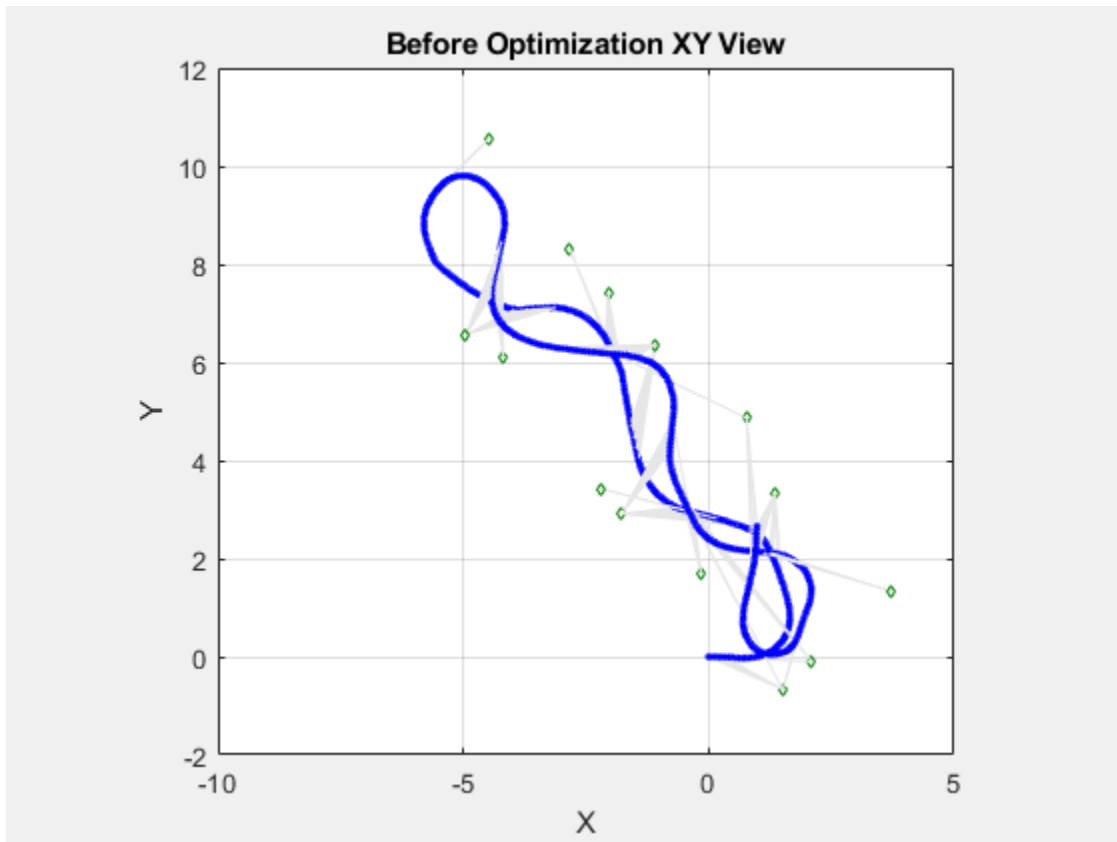
After building the pose graph, optimize it using the `optimizePoseGraph` function.

```
pgUpd = optimizePoseGraph(pg);
```

Visualize both the initial and optimized pose graph. The optimized pose graph shows these improvements:

- The initial position of the robot relative to the landmarks has been corrected.
- The landmarks on each wall are better aligned.
- The vertical drift in the robot trajectory along the z direction has been corrected.

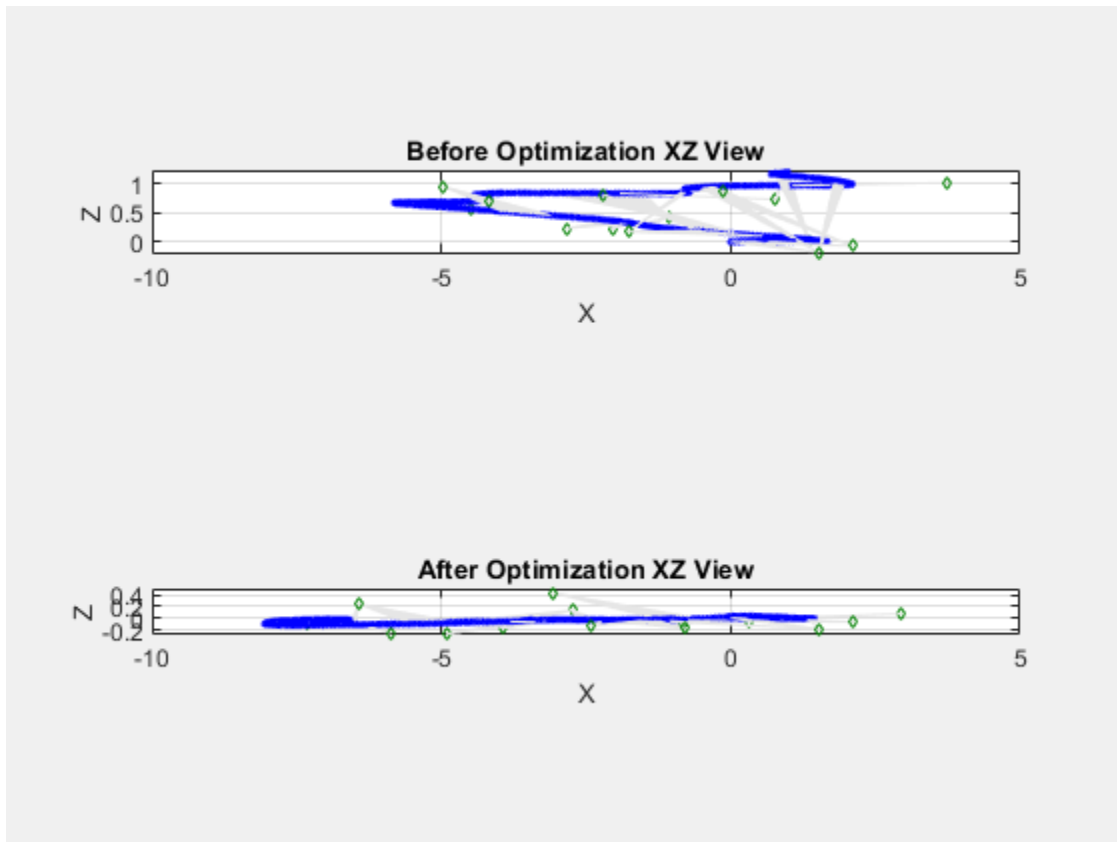
```
figure('Visible','on');  
show(pg); axis equal; xlim([-10.0 5.0]); ylim([-2.0 12.0]);  
title('Before Optimization XY View')
```



```
figure('Visible','on');  
show(pgUpd); axis equal; xlim([-10.0 5.0]); ylim([-2.0 12.0]);  
title('After Optimization XY View')
```

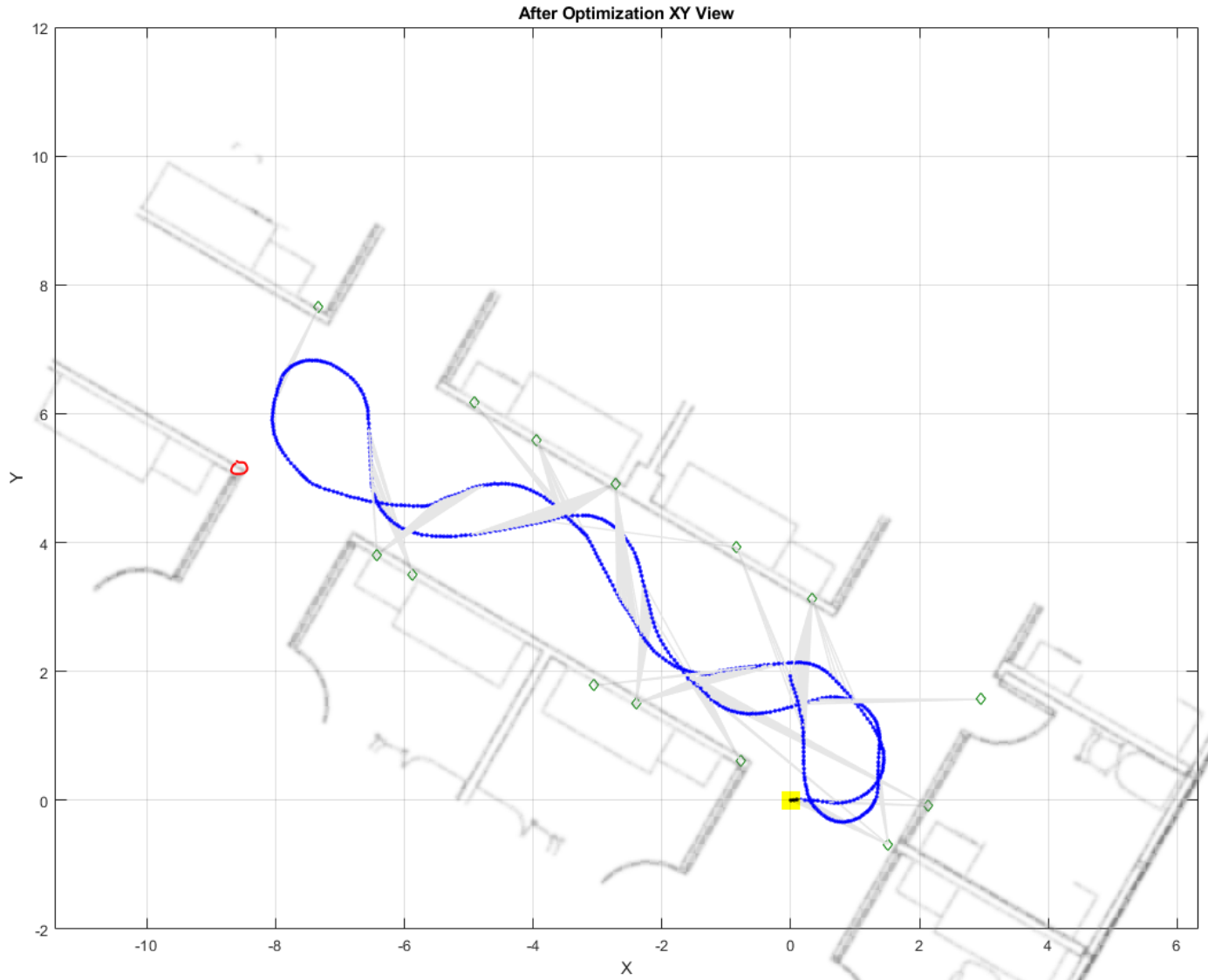


```
% Vertical drift.  
figure('Visible','on');  
subplot(2,1,1)  
show(pg); xlim([-10.0 5.0]); view(0, 0)  
title('Before Optimization XZ View')  
subplot(2,1,2)  
show(pgUpd); xlim([-10.0 5.0]); view(0, 0)  
title('After Optimization XZ View')
```

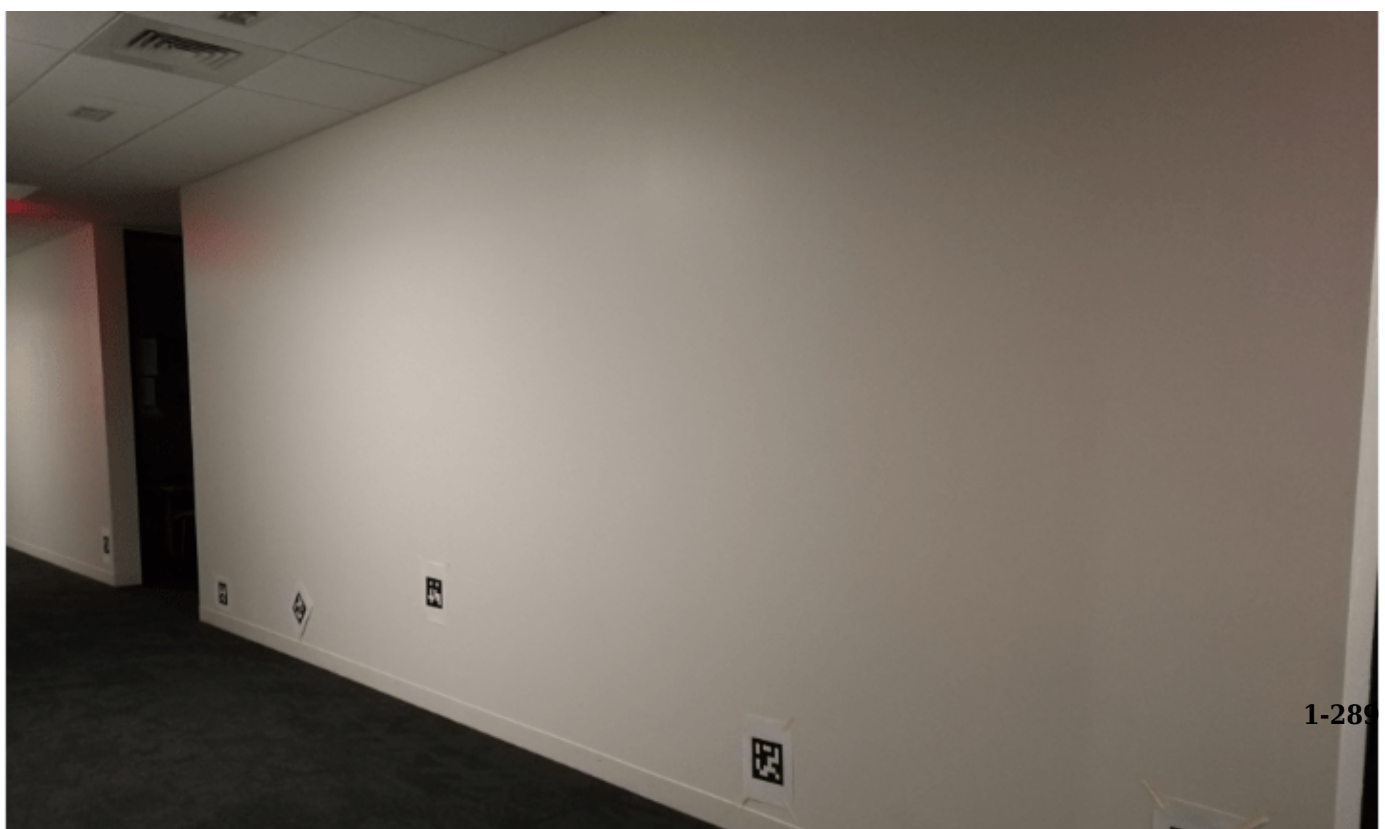


Conclusion

The final robot pose graph is overlaid on a blue print of the office area to show the true trajectory and the estimated landmark locations. If you playback the images again, you can see there is good correlation.



Note there is one AprilTag that is not picked up by the camera, marked by red circle in the following images. These images show the position of the AprilTags for your reference.





Extract Sensor Data from rosbag

The `exampleHelperExtractDataFromRosbag` function extracts synchronized image and odometry data from a rosbag recorded on a Jackal UGV robot (`apriltag_odometry_slam_dataset.bag` located in the same folder as the `.mat` file). Import from a rosbag (ROS Toolbox) requires ROS Toolbox.

```
function [imageData, odomData] = exampleHelperExtractDataFromRosbag(bagName)
%exampleHelperExtractDataFromRosbag Extract synchronized image and odometry
% data from a rosbag recorded on a Jackal UGV robot. ROS Toolbox license
% is required to use this function.

% Copyright 2020 The MathWorks, Inc.

bagSel = rosbag(bagName);
imgSel = bagSel.select('Topic','axis/image_raw_out/compressed'); % Image topic

% Needed for computing change between SE(2) poses
ss = stateSpaceSE2;
ss.WeightTheta = 1;

lastPoseSE2 = [];
lastT = [];

imageData = {};
odomData = {};

% Grab all the image messages
imgObjs = imgSel.readMessages;
for i = 1:numel(imgObjs)
```

```

% Odom data is extracted from tf
if bagSel.canTransform('odom', 'laser', imgObj{ i }.Header.Stamp) % From odom frame to laser
    % ith odom reading is extracted at the time of the ith image
    Tstamped = bagSel.getTransform('odom', 'laser', imgObj{ i }.Header.Stamp);
    [T, poseSE2] = translateTransformStampedMsg(Tstamped);

    if isempty(lastT)
        takeThisImage = true;
    else
        takeThisImage = true;
        % Only accept a new pair of sensor measurements if the robot odom pose
        % has changed more than this threshold since last accepted one
        % Here we use the stateSpaceSE2's distance function to calculate the 2D pose difference
        if ss.distance(poseSE2, lastPoseSE2) < 0.06
            takeThisImage = false;
        end
    end
end

if takeThisImage
    I = readImage(imgObj{ i }); % An alternative is to use the newer convenience function
    imageData{ end+1 } = I;
    odomData{ end+1 } = T;
    lastPoseSE2 = poseSE2;
    lastT = T;
end
end
end

function [ToutSE3, poseSE2] = translateTransformStampedMsg(Tin)
%translateTransformMsg Extract the 4x4 homogeneous transformation matrix,
% ToutSE3, from a TransformStamped message object. This function also
% returns a second output, poseSE2, which is an SE(2) pose vector
% computed by projecting ToutSE2 to the XY plane. Note the formulation
% used is approximate and relies on the assumption that the robot mostly
% moves on the flat ground.

% Copyright 2020 The MathWorks, Inc.

%Tin - TransformStamped message object
x = Tin.Transform.Translation.X;
y = Tin.Transform.Translation.Y;
z = Tin.Transform.Translation.Z;

qx = Tin.Transform.Rotation.X;
qy = Tin.Transform.Rotation.Y;
qz = Tin.Transform.Rotation.Z;
qw = Tin.Transform.Rotation.W;
q = [ qw, qx, qy, qz]; % Note the sequence for quaternion in MATLAB

ToutSE3 = robotics.core.internal.SEHelpers.poseToTformSE3([x,y,z,q]);
YPR = quat2eul(q);
poseSE2 = [x,y,YPR(1)];
end

```

Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs

This example shows how to reduce the drift in the estimated trajectory (location and orientation) of a monocular camera using 3-D pose graph optimization. Visual odometry estimates the current global pose of the camera (current frame). Because of poor matching or errors in 3-D point triangulation, robot trajectories often tends to drift from the ground truth. Loop closure detection and pose graph optimization reduce this drift and correct for errors.

Load Estimated Poses for Pose Graph Optimization

Load the estimated camera poses and loop closure edges. Estimated camera poses are computed using visual odometry. Loop closure edges are computed by finding previous frame which saw the current scene and estimating the relative pose between the current frame and the loop closure candidate. Camera frames are sampled from [1].

```
% Estimated poses
load('estimatedpose.mat');
% Loopclosure edge
load('loopedge.mat');
% Groundtruth camera locations
load('groundtruthlocations.mat');
```

Build 3-D Pose Graph

Create an empty pose graph.

```
pg3D = poseGraph3D;
```

Add nodes to the pose graph, with edges defining the relative pose and information matrix for the pose graph. Convert the estimated poses, given as transformations, to relative poses as an [x y theta qw qx qy qz] vector. An identity matrix is used for the information matrix for each pose.

```
len = size(estimatedPose,2);
informationmatrix = [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1];
% Insert relative poses between all successive frames
for k = 2:len
    % Relative pose between current and previous frame
    relativePose = estimatedPose{k-1}/estimatedPose{k};
    % Relative orientation represented in quaternions
    relativeQuat = tform2quat(relativePose);
    % Relative pose as [x y theta qw qx qy qz]
    relativePose = [tform2trvec(relativePose),relativeQuat];
    % Add pose to pose graph
    addRelativePose(pg3D,relativePose,informationmatrix);
end
```

Add a loop closure edge. Add this edge between two existing nodes from the current frame to a previous frame.

```
% Convert pose from transformation to pose vector.
relativeQuat = tform2quat(loopedge);
relativePose = [tform2trvec(loopedge),relativeQuat];
% Loop candidate
loopcandidateframeid = 1;
% Current frame
```

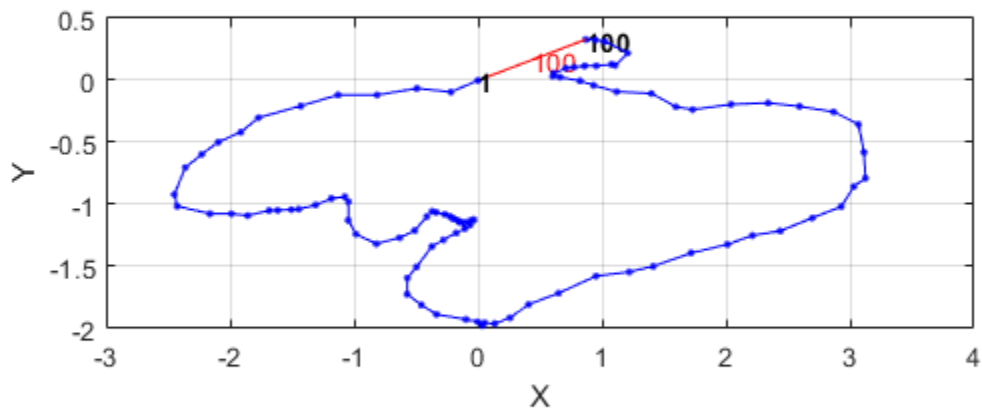
```

currentframeid = 100;

addRelativePose(pg3D,relativePose,informationmatrix,...
               loopcandidateframeid,currentframeid);

figure
show(pg3D);

```

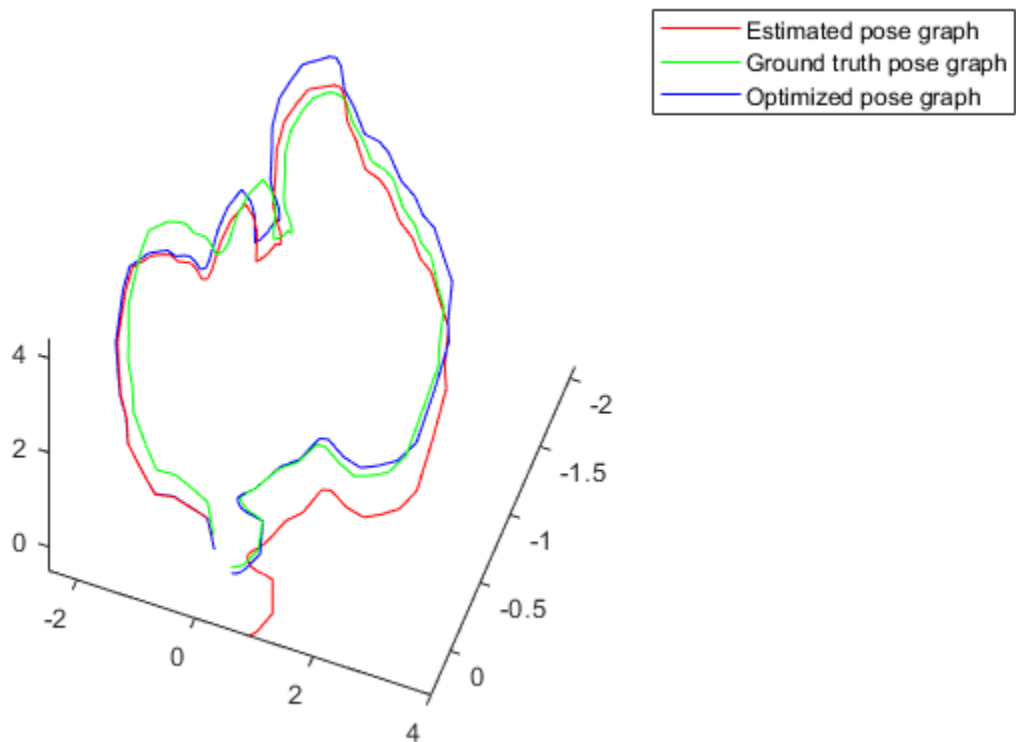


Optimize the pose graph. The nodes are adjusted based on the edge constraints to improve the overall pose graph. To see the change in drift, plot the estimated poses and the new optimized poses against the ground truth.

```

% Pose graph optimization
optimizedPosegraph = optimizePoseGraph(pg3D);
optimizedposes = nodes(optimizedPosegraph);
% Camera trajectory plots
figure
estimatedposes = nodes(pg3D);
plot3(estimatedposes(:,1),estimatedposes(:,2),estimatedposes(:,3),'r');
hold on
plot3(groundtruthlocations(:,1),groundtruthlocations(:,2),groundtruthlocations(:,3),'g');
plot3(optimizedposes(:,1),optimizedposes(:,2),optimizedposes(:,3),'b');
hold off
legend('Estimated pose graph','Ground truth pose graph', 'Optimized pose graph');
view(-20.8,-56.4);

```



References

- [1] Galvez-López, D., and J. D. Tardós. "Bags of Binary Words for Fast Place Recognition in Image Sequences." *IEEE Transactions on Robotics*. Vol. 28, No. 5, 2012, pp. 1188-1197.

Create Egocentric Occupancy Maps Using Range Sensors

Occupancy Maps offer a simple yet robust way of representing an environment for robotic applications by mapping the continuous world-space to a discrete data structure. Individual *grid cells* can contain binary or probabilistic information, where 0 indicates free-space, and 1 indicates occupied space. You can build up this information over time using sensor measurements and efficiently store them in the map. This information is also useful for more advanced workflows, such as collision detection and path planning.

This example shows how to create an *egocentric* occupancy map, which keeps track of the immediate surroundings of the robot which can be efficiently moved around the environment. A trajectory is generated by planning a path in the environment and dictates the motion of the robot. As the robot moves around, the map is updated using sensor information from a simulated lidar and ground-truth map.

Load a Prebuilt Ground-Truth Occupancy Map

Create a non-egocentric map from a previously generated data file, which is considered to be the ground truth for the simulated lidar. Load the map, `mapData`, which contains the `Data` field as a probabilistic matrix and convert it to binary values.

Create a `binaryOccupancyMap` object with the binary matrix and specify the resolution of the map.

```
% Load saved map information
load mapData_rayTracingTrajectory
binaryMatrix = mapData.Data > 0.5;
worldMap = binaryOccupancyMap(binaryMatrix,mapData.Resolution);
```

Set the location of the bottom-left corner of the map as the world origin.

```
worldMap.LocalOriginInWorld = mapData.GridLocationInWorld;
```

Plot the ground truth. This example sets up a subplot for showing two maps side by side.

```
set(gcf,'Visible','on')
worldAx = subplot(1,2,1);
worldHandle = show(worldMap,'Parent',worldAx);
hold all
```

Create a Simulated Lidar

Create a `rangeSensor` object, which can be used to gather lidar readings from the simulation. You can modify various properties on the `rangeSensor` to more accurately represent a particular model of lidar, or add in sensor noise to test the robustness of your solution. For this example, set the `[min max]` range and the noise parameter. After creating the object, retrieve and plot a reading from the sensor by providing an `[x y theta]` pose relative to the world frame. Example helpers plot the robot, and the lidar readings overtop the `worldMap`.

```
% Create rangeSensor
lidar = rangeSensor;
lidar.Range = [0 5];
lidar.RangeNoise = 0;
pos = [0 0 0];

% Show lidar readings in world map
```

```
[ranges, angles] = lidar(pos, worldMap);
hSensorData = exampleHelperPlotLidar(worldAx, pos, ranges, angles);

% Show robot in world map
hRobot = exampleHelperPlotRobot(worldAx, pos);
```



Initialize an Egocentric Map

Create an `occupancyMap` object to represent the egocentric map. Set the local-origin to the center location of the grid. Shift the grid origin by half the width of the bounds.

```
% By default, GridOriginInLocal = [0 0]
egoMap = occupancyMap(10,10,worldMap.Resolution);

% Offset the GridOriginInLocal such that the "local frame" is located in the
% center of the "map-window"
egoMap.GridOriginInLocal = -[diff(egoMap.XWorldLimits) diff(egoMap.YWorldLimits)]/2;
```

Plot the egocentric map next to the world map in the subplot.

```
% Update local plot
localAx = subplot(1,2,2);
show(egoMap, 'Parent', localAx);
hold all
localMapFig = plot(localAx, egoMap.LocalOriginInWorld+[0 1], egoMap.LocalOriginInWorld+[0 0], 'r-')
```


Plan Path Between Points

We can now use our ground-truth map to plan a path between two free points. Create a copy of the world map and inflate it based on the robot size and desired clearance. This example uses a car-like robot, which has non-holonomic constraints, specified with a `stateSpaceDubins` object. This state space is used by the path planner for randomly sampling feasible states for the robot. Lastly, create a `validatorOccupancyMap` object, which uses the map to validate generated states and the motions that connect them by checking the corresponding cells for occupancy.

```
% Copy the world map and inflate it.
binaryMap = binaryOccupancyMap(worldMap);
inflate(binaryMap, 0.1);

% Create a state space object.
stateSpace = stateSpaceDubins;

% Reduce the turning radius to better fit the size of map and obstacle
% density.
stateSpace.MinTurningRadius = 0.5;

% Create a state validator object.
validator = validatorOccupancyMap(stateSpace);
validator.Map = binaryMap;
validator.ValidationDistance = 0.1;
```

Use the RRT* planning algorithm as a `plannerRRTStar` object and specify the state space and state validator as inputs. Specify start and end locations for the planner and generate a path.

```
% Create our planner using the previously created StateSpace and
% StateValidator objects.
planner = plannerRRTStar(stateSpace, validator);
planner.MaxConnectionDistance = 2;
planner.MaxIterations = 20000;

% Set a seed for the randomly generated path for reproducible results.
rng(1, 'twister')

% Set the start and end points.
startPt = [-6 -5 0];
goalPt = [ 8 7 pi/2];

% Plan a path between start and goal points.
path = plan(planner, startPt, goalPt);
interpolate(path, size(path.States,1)*10);
plot(worldAx, path.States(:,1),path.States(:,2), 'b-');
```

Generate a Trajectory Along the Path

The planner generated a set of states, but to execute a trajectory, times for the states are needed. The goal of this example is to move the robot along the path with a constant linear velocity of 0.5 m/s. To get timestamps for each point, calculate the distances between points, sum them cumulatively, then divide this by the linear velocity to get a monotonically increasing array of timestamps, `tStamps`.

```
% Get distance between each waypoint
pt2ptDist = distance(stateSpace,path.States(1:end-1,:),path.States(2:end,:))

pt2ptDist = 129x1
```



```

% Retrieve sensor information from the lidar and insert it into the egoMap
[ranges, angles] = lidar(robotCurrentPose, worldMap);
insertRay(egoMap, robotCurrentPose, ranges, angles, lidar.Range(2));

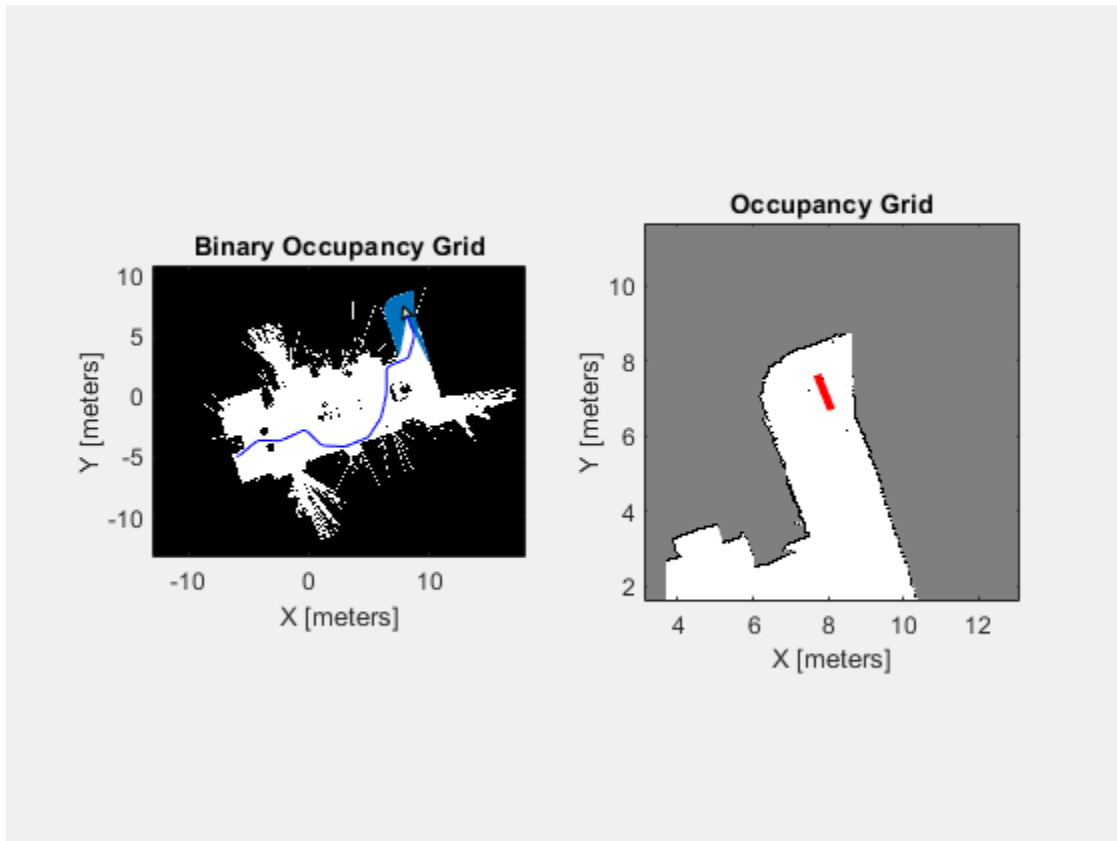
% Update egoMap-centric plot
show(egoMap, 'Parent', localAx, 'FastUpdate', 1);

% Update orientation vector
set(localMapFig, 'XData', robotCurrentPose(1)+[0 cos(robotCurrentPose(3))], 'YData', robotCurrentPose(2)+[0 sin(robotCurrentPose(3))]);

% Update world plot
exampleHelperUpdateRobotAndLidar(hRobot, hSensorData, robotCurrentPose, ranges, angles);

% Call drawnow to push updates to the figure
drawnow limitrate
end

```



Build Occupancy Map from Lidar Scans and Poses

The `buildMap` function takes in lidar scan readings and associated poses to build an occupancy grid as `lidarScan` objects and associated `[x y theta]` poses to build an `occupancyMap`.

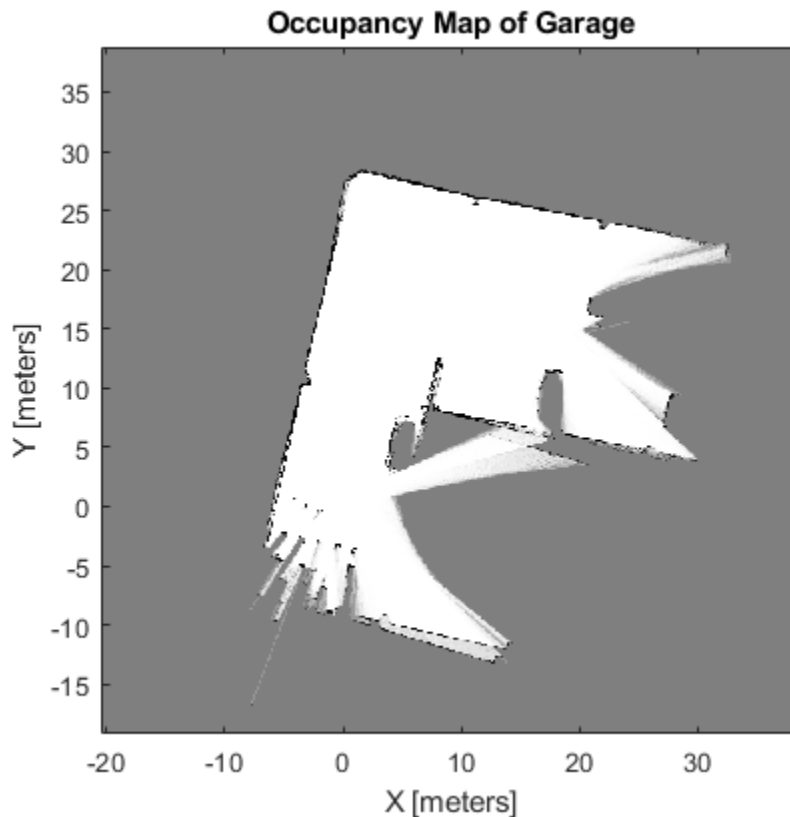
Load scan and pose estimates collected from sensors on a robot in a parking garage. The data collected is correlated using a `lidarSLAM` algorithm, which performs scan matching to associate scans and adjust poses over the full robot trajectory. Check to make sure scans and poses are the same length.

```
load scansAndPoses.mat
length(scans) == length(poses)

ans = logical
     1
```

Build the map. Specify the scans and poses in the `buildMap` function and include the desired map resolution (10 cells per meter) and the max range of the lidar (19.2 meters). Each scan is added at the associated poses and probability values in the occupancy grid are updated.

```
occMap = buildMap(scans,poses,10,19.2);
figure
show(occMap)
title('Occupancy Map of Garage')
```



Create Egocentric Occupancy Map from Driving Scenario Designer

This example shows how to create an egocentric occupancy map from the **Driving Scenario Designer app**. This example uses obstacle information from the vision detection generator to update the egocentric occupancy map.

This example:

- Gets obstacle information and road geometry using vision detection generator.
- Creates an ego centric occupancy map using binary occupancy map.
- Updates the ego centric occupancy map using lane boundaries and obstacle information.

Introduction

Automated driving systems use multiple on-board sensors on the ego vehicle like radar, cameras, and lidar. These sensors are used to perceive information from the surroundings and environment. It is important to collate information from these heterogeneous sensors into a common temporal frame of reference. This is usually done using an egocentric occupancy map. This map contains information about the surrounding environment like road geometry, free space, and obstacles. This egocentric occupancy map is used by planning algorithms for navigation. The ego vehicle can respond to dynamic changes in the environment by periodically updating the information in this ego centric occupancy map.

This example shows how to use lane and obstacle information obtained from a scenario to create and update an ego centric occupancy map.

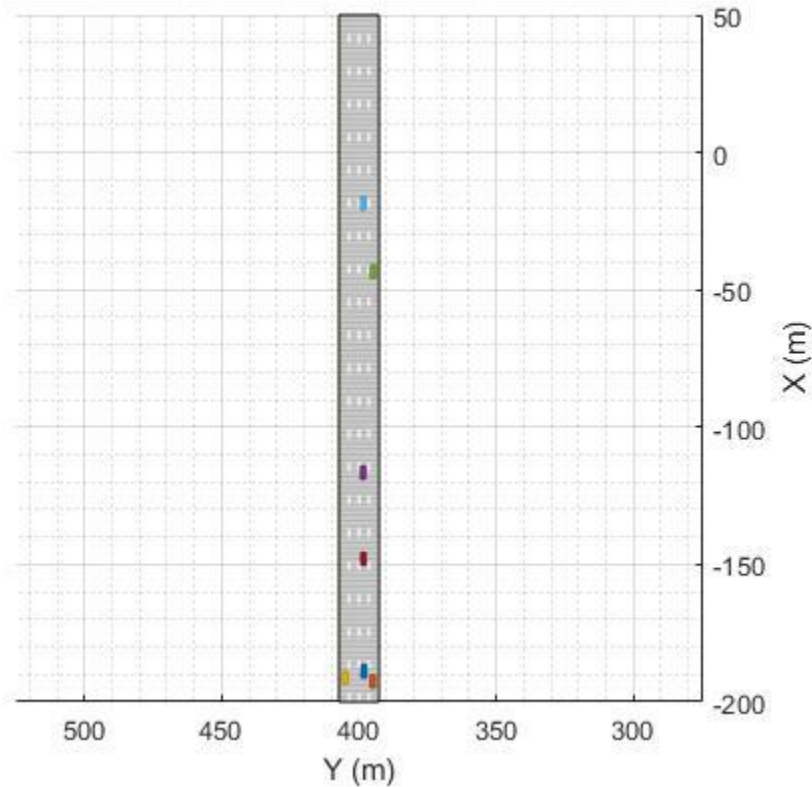
This example also uses a straight road scenario designed using Driving Scenario Designer (DSD). For more details, see Driving Scenario Designer (Automated Driving Toolbox). You can also create a Driving Scenario programmatically. For more details, refer to “Create Driving Scenario Programmatically” (Automated Driving Toolbox).

Get Lane Boundaries and Obstacle Information from Driving Scenario

The scenario used in this example is a straight road with four lanes. This scenario has one ego vehicle and six target vehicles which follow their respective predefined paths. Create a scenario using the helper function, `exampleHelperCreateStraightRoadScenario`.

The scenario used in this example is shown in the following figure.

```
[scenario, egoVehicle] = exampleHelperCreateStraightRoadScenario;
```



This example uses a vision detection generator which synthesizes camera sensor mounted at the front of the ego vehicle in the driving scenario. This generator is configured to detect lane boundaries and obstacles.

For more details, see `visionDetectionGenerator` (Automated Driving Toolbox).

The update interval of the vision detection generator is configured to generate detections at 0.1 second intervals, which is consistent with the update rate of typical automotive vision sensors.

Create the `actorProfiles` for the scenario, which contain physical and radar profiles of all driving scenario actors including the ego vehicle. Specify those actors and other configuration parameters for the vision detection generator.

```
profiles = actorProfiles(scenario);

% Configure vision detection generator to detect lanes and obstacles
sensor = visionDetectionGenerator('SensorIndex', 1, ...
    'SensorLocation', [1.9 0], ...
    'DetectionProbability', 1, ...
    'MinObjectImageSize', [5 5], ...
    'FalsePositivesPerImage', 0, ...
    'DetectorOutput', 'Lanes and objects', ...
    'Intrinsics', cameraIntrinsics([700 1814],[320 240],[480 640]), ...
    'ActorProfiles', profiles,...
    'UpdateInterval', 0.1);
```

To get detections, the sensor object is called for each simulation step. This sensor object call occurs in the helper function `exampleHelperGetObstacleDataFromSensor`.

Create an Egocentric Occupancy Map

This example uses a `binaryOccupancyMap` object to create an egocentric occupancy map.

Create a square occupancy map with 100 meters per side and a resolution of 2 cells per meter. Set all the cells to occupied state by default.

```
egoMap = binaryOccupancyMap(100, 100, 2);
setOccupancy(egoMap, ones(200, 200));
```

By default, the map origin is at bottom-left. Move the `egoMap` origin to the center of the occupancy map. This converts the occupancy map to an egocentric occupancy map.

```
egoMap.GridOriginInLocal = [-egoMap.XLocalLimits(2)/2, ...
                           -egoMap.YLocalLimits(2)/2];
```

Update the Egocentric Occupancy Map with Obstacle Information

Before updating the `egoMap`, initialize the visualization window. Obstacles in the `egoMap` visualization are represented as black (occupied) and free space as white (unoccupied).

```
hAxes = exampleHelperSetupVisualization(scenario);
```

Setup a loop for executing the scenario using `advance(scenario)`. This loop should move the `egoVehicle` along the road, updating the pose as it moves.

Call `move` on the `egoMap` using the updated pose to get updated detections. Clear the map of all obstacles for each update.

```
% Advancing the scenario
while advance(scenario)
    % Ego vehicle position
    egoPose = egoVehicle.Position;
    egoYaw = deg2rad(egoVehicle.Yaw);

    % Move the origin of grid to the face of the ego vehicle.
    move(egoMap, [egoPose(1), egoPose(2)]);

    %Reset the egoMap before updating with obstacle information
    setOccupancy(egoMap, ones(egoMap.GridSize));
```

Lane boundaries and obstacles information generated from a vision detection generator are used to find occupied space that needs to be updated in the `egoMap`.

This example uses road boundaries information extracted from the lane boundaries. The region outside road boundaries is also considered occupied.

The `exampleHelperGetObstacleDataFromSensor` helper function extracts road boundaries and obstacle information generated from the vision detection generator.

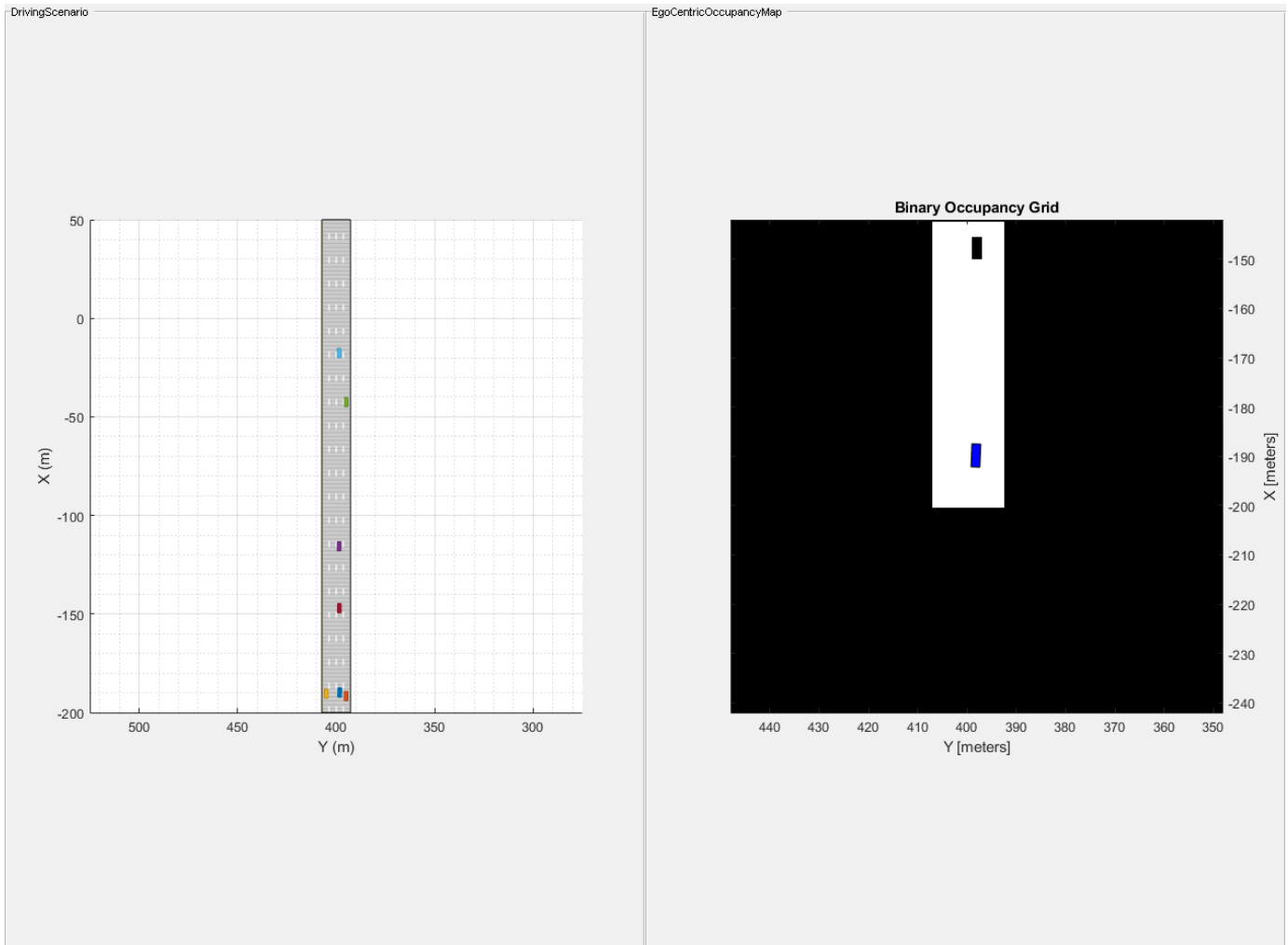
```
[obstacleInfo, roadBorders, isValidLaneTime] = ...
    exampleHelperGetObstacleDataFromSensor(scenario, egoMap, ...
                                           egoVehicle, sensor);
```

Depending upon the range of the vision detection generator, the detections may fall outside the egoMap bounds. Using the egoMap bounds, the exampleHelperFilterObstacles function extracts the obstacles and free space that are within the egocentric map.

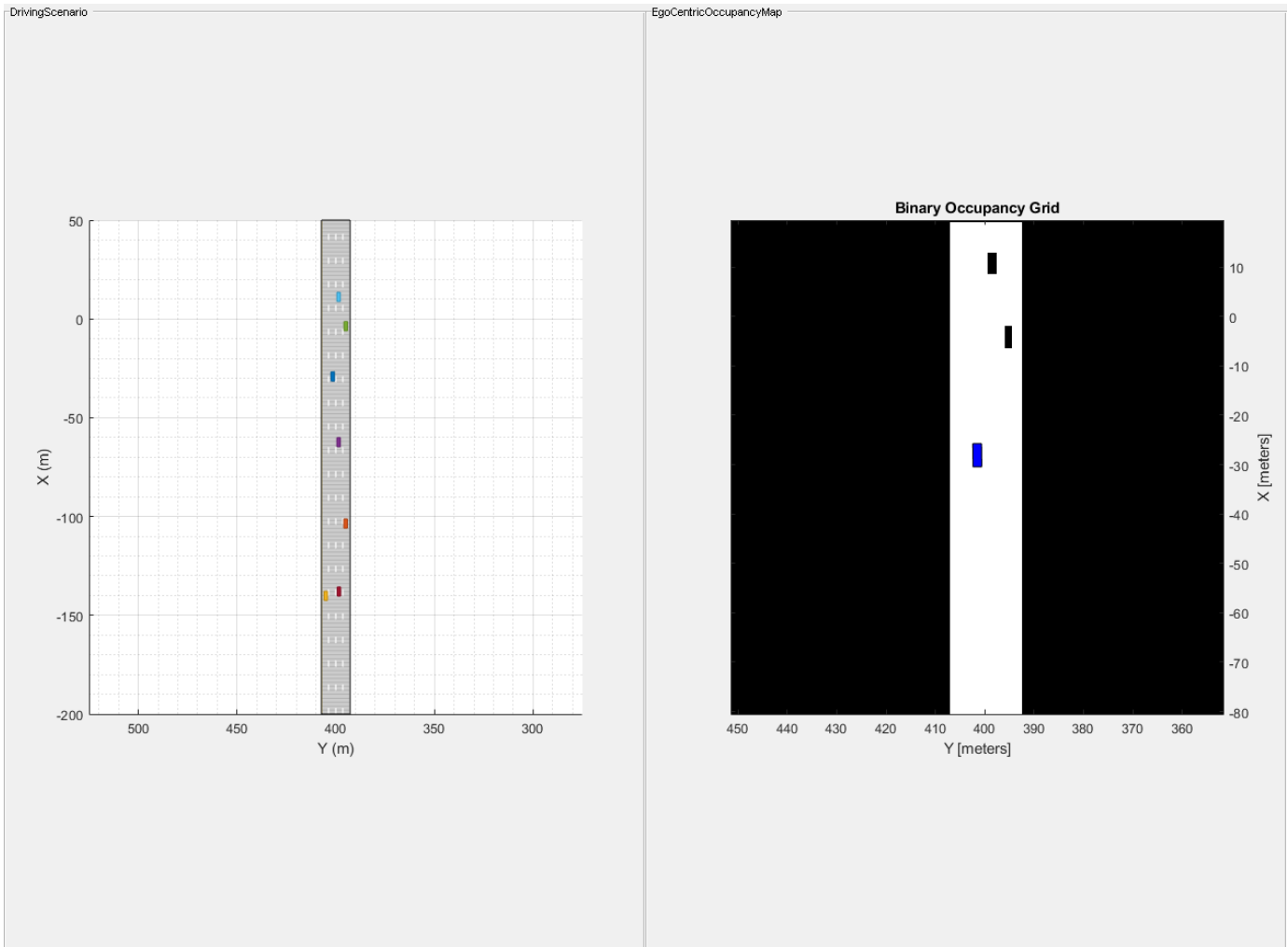
```
[obstaclePoints, unoccupiedSpace] = exampleHelperFilterObstacles(...  
    egoMap, obstacleInfo, ...  
    roadBorders, ...  
    isValidLaneTime, egoVehicle);
```

Use the filtered obstacle and free space locations to update the egocentric map. Update the visualization.

```
% Set the occupancy of free space to 0  
if ~isempty(unoccupiedSpace)  
    setOccupancy(egoMap, unoccupiedSpace, 0);  
end  
  
% Set the occupancy of occupied space to 1  
if ~isempty(obstaclePoints)  
    setOccupancy(egoMap, obstaclePoints, 1);  
end  
  
% Updating the visualization  
exampleHelperUpdateVisualization(hAxes, egoVehicle, egoPose, egoYaw, egoMap);  
end
```

1 Navigation Featured Examples



Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph

This example shows how to reduce the drift in the estimated trajectory (location and orientation) of a monocular camera using 3-D pose graph optimization. In this example, you build an occupancy map from the depth images, which can be used for path planning while navigating in that environment.

Load Estimated Poses for Pose Graph Optimization

Load the estimated camera poses and loop closure edges. The estimated camera poses were computed using visual odometry. The loop closure edges were computed by finding the previous frame that saw the current scene and estimating the relative pose between the current frame and the loop closure candidate. Camera frames are sampled from a data set that contains depth images, camera poses, and ground truth locations [1].

```
load('estimatedpose.mat');           % Estimated poses
load('loopedge.mat');               % Loopclosure edge
load('groundtruthlocations.mat');   % Ground truth camera locations
```

Build 3-D Pose Graph

Create an empty pose graph.

```
pg3D = poseGraph3D;
```

Add nodes to the pose graph, with edges defining the relative pose and information matrix for the pose graph. Convert the estimated poses, given as transformations, to relative poses as an $[x \ y \ \theta \ q_w \ q_x \ q_y \ q_z]$ vector. An identity matrix is used for the information matrix for each pose.

```
len = size(estimatedPose,2);
informationmatrix = [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1];
% Insert relative poses between all successive frames
for k = 2:len
    % Relative pose between current and previous frame
    relativePose = estimatedPose{k-1}/estimatedPose{k};
    % Relative orientation represented in quaternions
    relativeQuat = tform2quat(relativePose);
    % Relative pose as [x y theta qw qx qy qz]
    relativePose = [tform2trvec(relativePose),relativeQuat];
    % Add pose to pose graph
    addRelativePose(pg3D,relativePose,informationmatrix);
end
```

Add a loop closure edge. Add this edge between two existing nodes from the current frame to a previous frame. Optimize the pose graph to adjust nodes based on the edge constraints and this loop closure. Store the optimized poses.

```
% Convert pose from transformation to pose vector.
relativeQuat = tform2quat(loopedge);
relativePose = [tform2trvec(loopedge),relativeQuat];
% Loop candidate
loopcandidateframeid = 1;
% Current frame
currentframeid = 100;

addRelativePose(pg3D,relativePose,informationmatrix,...
```

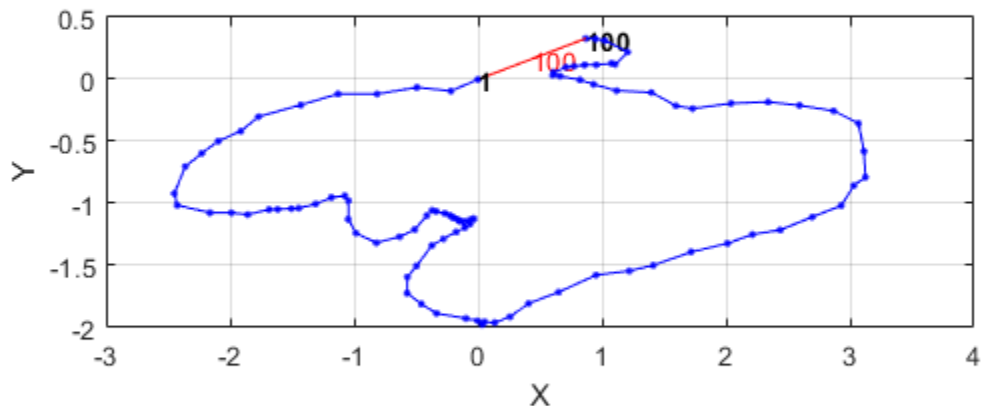
```

        loopcandidateframeid,currentframeid);

optimizedPosegraph = optimizePoseGraph(pg3D);
optimizedposes = nodes(optimizedPosegraph);

figure;
show(pg3D);

```



Create Occupancy Map from Depth Images and Optimized Poses

Load the depth images and camera parameters from the dataset [1].

```

load('depthimagearray.mat'); % variable depthImages
load('freburgK.mat'); % variable K

```

Create a 3-D occupancy map with a resolution of 50 cells per meter. Read in the depth images iteratively and convert the points in the depth image using the camera parameters and the optimized poses of the camera. Insert the points as point clouds at the optimized poses to build the map. Show the map after adding all the points. Because there are many depth images, this step can take several minutes. Consider uncommenting the `fprintf` command to print the progress the image processing.

```

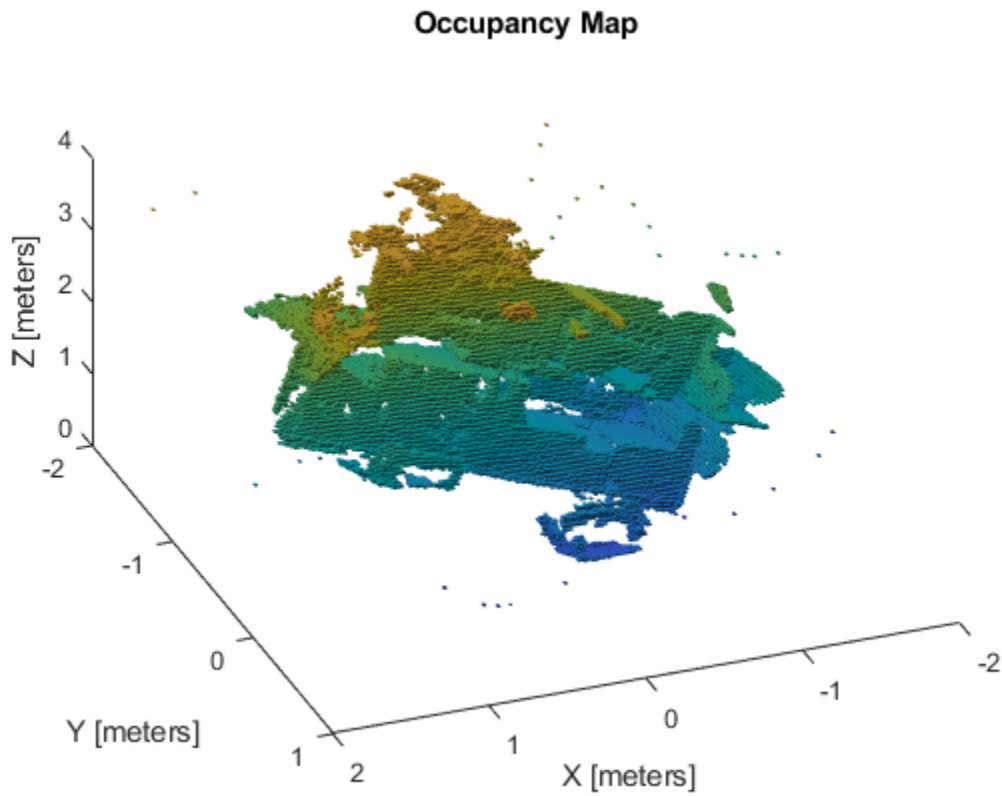
map3D = occupancyMap3D(50);

for k = 1:length(depthImages)
    points3D = exampleHelperExtract3DPointsFromDepthImage(depthImages{k},K);
    % fprintf('Processing Image %d\n', k);
    insertPointCloud(map3D,optimizedposes(k,:),points3D,1.5);
end

```

Show the map.

```
figure;  
show(map3D);  
xlim([-2 2])  
ylim([-2 1])  
zlim([0 4])  
view([-201 47])
```



References

[1] Galvez-López, D., and J. D. Tardós. "Bags of Binary Words for Fast Place Recognition in Image Sequences." *IEEE Transactions on Robotics*. Vol. 28, No. 5, 2012, pp. 1188-1197.

Fuse Multiple Lidar Sensors Using Map Layers

Occupancy maps offer a simple yet robust way of representing an environment for robotic applications by mapping the continuous world-space to a discrete data structure. Individual *grid cells* can contain binary or probabilistic information about obstacle information. However, an autonomous platform may use a variety of sensors that may need to be combined to estimate both the current state of the platform and the state of the surrounding environment.

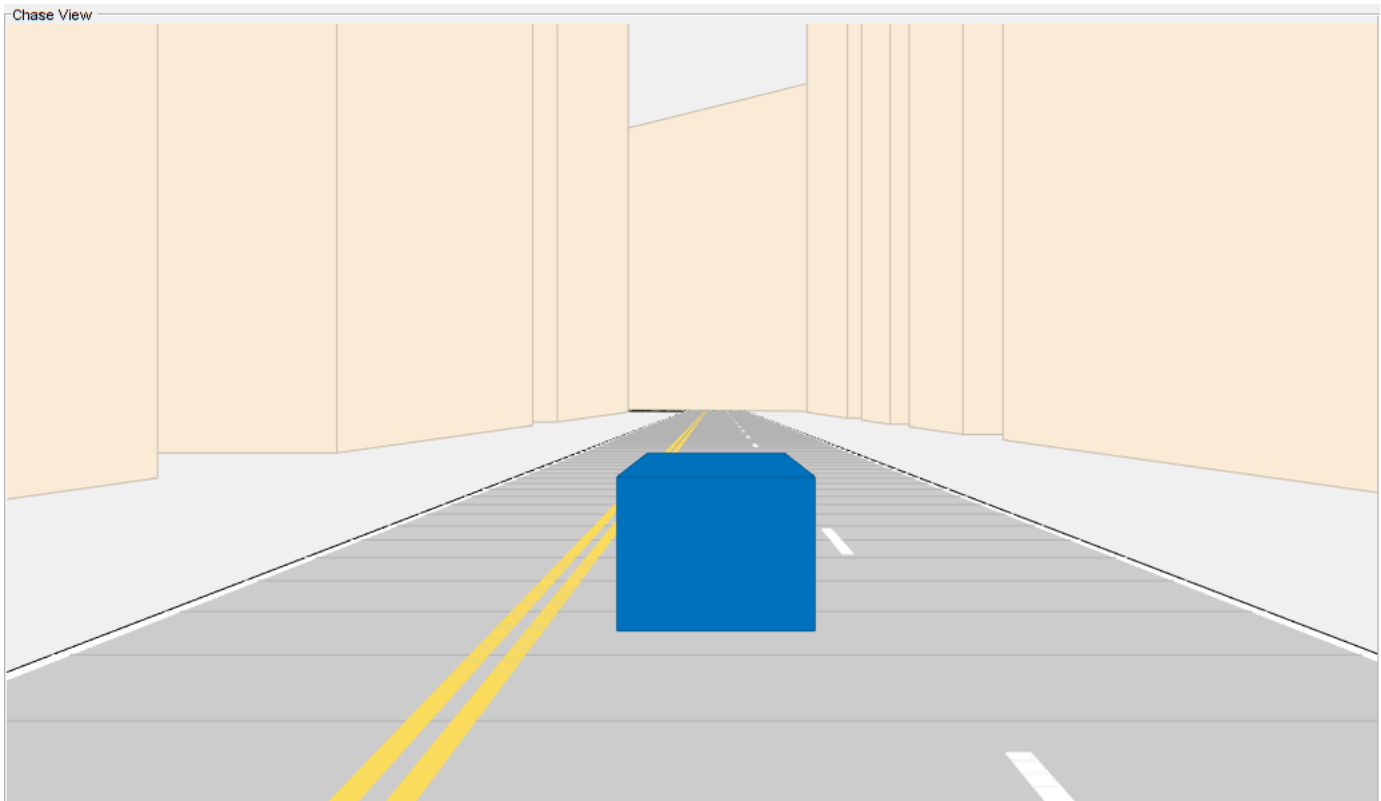
This example focuses on integrating a variety of sensors to estimate the state of the environment and store occupancy values are different map layers. The example shows how the `multiLayerMap` object can be used to visualize, debug, and fuse data gathered from three lidar sensors mounted to an autonomous vehicle. The sensor readings in this example are simulated using a set of `lidarPointCloudGenerator` (Automated Driving Toolbox) objects which capture readings from the accompanying `drivingScenario` (Automated Driving Toolbox) object.

Each lidar updates its own `validatorOccupancyMap3D` object which enables us to visualize the local map created by each sensor in isolation. These local maps can be used to quickly identify sources of noise or mounting error, and can help in choosing an appropriate fusion technique. The `multiLayerMap` contains a fourth `mapLayer` object, which uses a custom callback function to fuse the data contained in each occupancy layer. Lastly, the fused map is used to update the corresponding subregion of a world map as the autonomous vehicle moves along the preplanned path.

Load Driving Scenario

First, create a `drivingScenario` object and populate the scene with several buildings using an example helper function. The function also visualizes the scene.

```
scene = drivingScenario;  
groundTruthVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);  
  
% Add a road and buildings to scene and visualize.  
exampleHelperPopulateScene(scene, groundTruthVehicle);
```



Generate a trajectory that follows the main road in the scene using a `waypointTrajectory` object.

```

sampleRate = 100;
speed = 10;
t = [0 20 25 44 46 50 54 56 59 63 90].';
wayPoints = [ 0 0 0;
              200 0 0;
              200 50 0;
              200 230 0;
              215 245 0;
              260 245 0;
              290 240 0;
              310 258 0;
              290 275 0;
              260 260 0;
              -15 260 0];
velocities = [ speed 0 0;
               speed 0 0;
               0 speed 0;
               0 speed 0;
               speed 0 0;
               speed 0 0;
               speed 0 0;
               0 speed 0;
               -speed 0 0;
               -speed 0 0;
               -speed 0 0];

```

```
traj = waypointTrajectory(wayPoints, 'TimeOfArrival', t, ...
    'Velocities', velocities, 'SampleRate', sampleRate);
```

Create Simulated Lidar Sensors

To gather lidar readings from the driving scenario, create three `lidarPointCloudGenerator` objects using an example helper function. This vehicle has been configured to have two front-facing, narrow field-of-view (FOV) lidars and a single wide FOV rear-facing Lidar. The overlapping region of both front-facing sensors should help to quickly register and confirm free space ahead of the vehicle, whereas the rear-facing sensor range helps map the traversed region.

```
lidarSensors = exampleHelperCreateVehicleSensors(scene, groundTruthVehicle);
disp(lidarSensors)
```

```
    {1x1 lidarPointCloudGenerator}    {1x1 lidarPointCloudGenerator}    {1x1 lidarPointCloudGener
```

Initialize Egocentric Map

Create a `multiLayerMap` object composed of three `occupancyMap` objects and one generic `mapLayer` object. Each local `occupancyMap` is updated by the corresponding lidar sensor. To combine data from all maps into the `mapLayer` object, set the `GetTransformFcn` name-value argument to the `exampleHelperFuseOnGet` function stored as a handle `fGet`. The `exampleHelperFuseOnGet` function fused all three maps data values by calling the `getMapData` function on each and using a log-odds summation of the values.

```
% Define map and parameters.
```

```
res = 2;
width = 100*2;
height = 100*2;
```

```
% Define equal weights for all sensor readings.
```

```
weights = [1 1 1];
```

```
% Create mapLayers for each sensor.
```

```
fLeftLayer = occupancyMap(width,height,res, 'LayerName', 'FrontLeft');
fRightLayer = occupancyMap(width,height,res, 'LayerName', 'FrontRight');
rearLayer = occupancyMap(width,height,res, 'LayerName', 'Rear');
```

```
% Create a get callback used to fuse data in the three layers.
```

```
fGet = @(obj,values,varargin)...
    exampleHelperFuseOnGet(fLeftLayer,fRightLayer,rearLayer,...
        weights,obj,values,varargin{:});
```

```
% Create a generic mapLayer object whose getMapData function fuses data from all
% three layers.
```

```
fusedLayer = mapLayer(width,height, 'Resolution', res, 'LayerName', 'FuseLayer', ...
    'GetTransformFcn', fGet, 'DefaultValue', 0.5);
```

```
% Combine layers into a multiLayerMap.
```

```
egoMap = multiLayerMap({fLeftLayer, fRightLayer, rearLayer, fusedLayer});
```

```
% Set map grid origin so that the robot is located at the center.
```

```
egoMap.GridOriginInLocal = -[diff(egoMap.XLocalLimits) diff(egoMap.YLocalLimits)]/2;
```

Create Reconstruction Map

Create an empty world map. This map is periodically updated using data from the fusion layer. Use this map to indicate how well the lidar fusion method is working.


```
% Create an empty reconstruction layer covering the same area as world map.
reconLayer = occupancyMap(400,400,res,... % width,height,resolution
    'LayerName','FuseLayer','LocalOriginInWorld',[-25 -50]);
```

Setup Visualization

Plot the egocentric layers next to the reconstructed map. Use the `exampleHelperShowEgoMap` function to display each local map.

```
% Setup the display window.
```

```
axList = exampleHelperSetupDisplay(groundTruthVehicle,lidarSensors);
```

```
% Display the reconstructionLayer and submap region.
```

```
show(reconLayer,'Parent', axList{1});
```

```
hG = findobj(axList{1},'Type','hggroup');
```

```
egoOrientation = hG.Children;
```

```
egoCenter = hgtransform('Parent',hG);
```

```
egoOrientation.Parent = egoCenter;
```

```
gridLoc = egoMap.GridLocationInWorld;
```

```
xLimits = egoMap.XLocalLimits;
```

```
yLimits = egoMap.YLocalLimits;
```

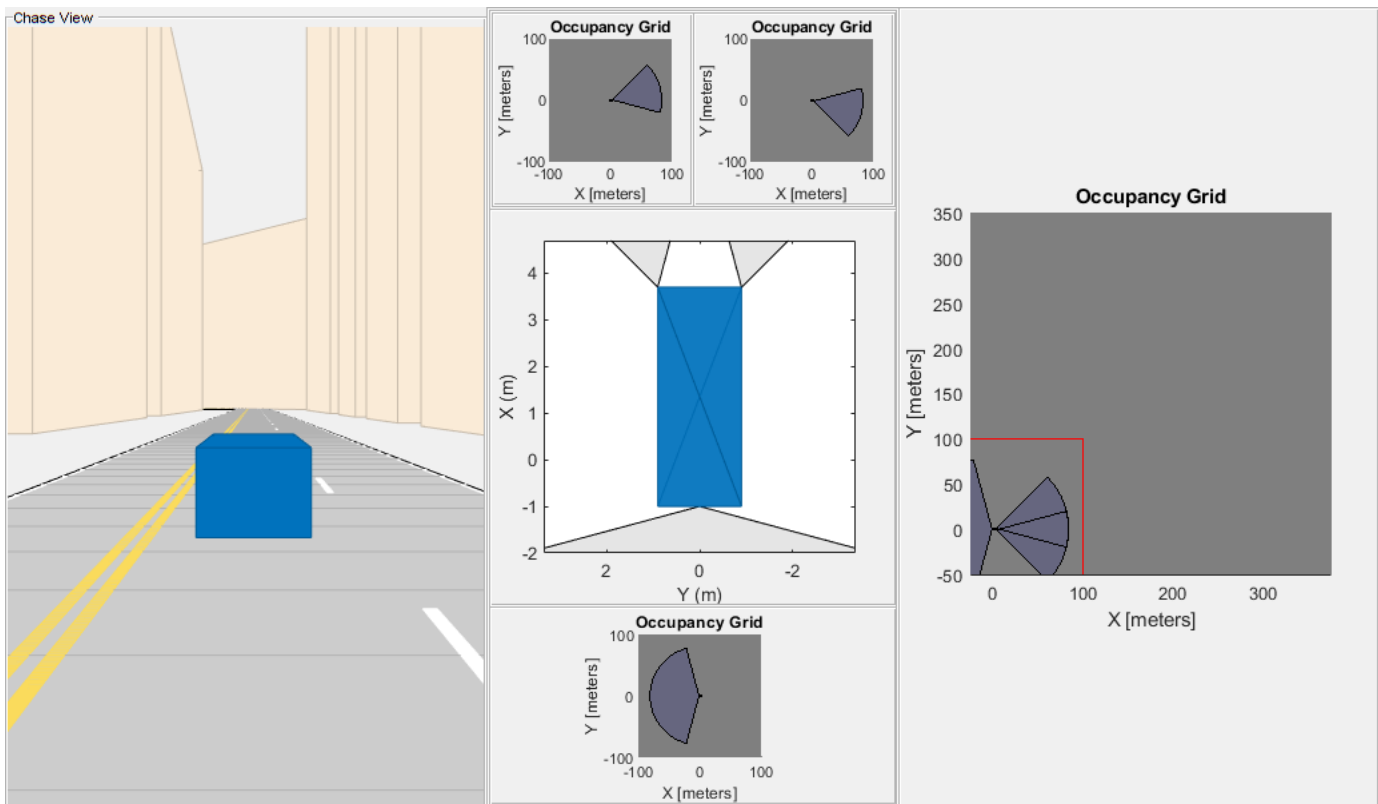
```
rectangle('Parent',egoCenter,...
```

```
    'Position',[gridLoc diff(xLimits) diff(yLimits)],...
```

```
    'EdgeColor','r');
```

```
% Display the local maps built by each sensor alongside the reconstruction map.
```

```
exampleHelperShowEgoMap(axList,egoMap,[0 0 0],{'FrontLeft Lidar','FrontRight Lidar','Rear Lidar'}
```

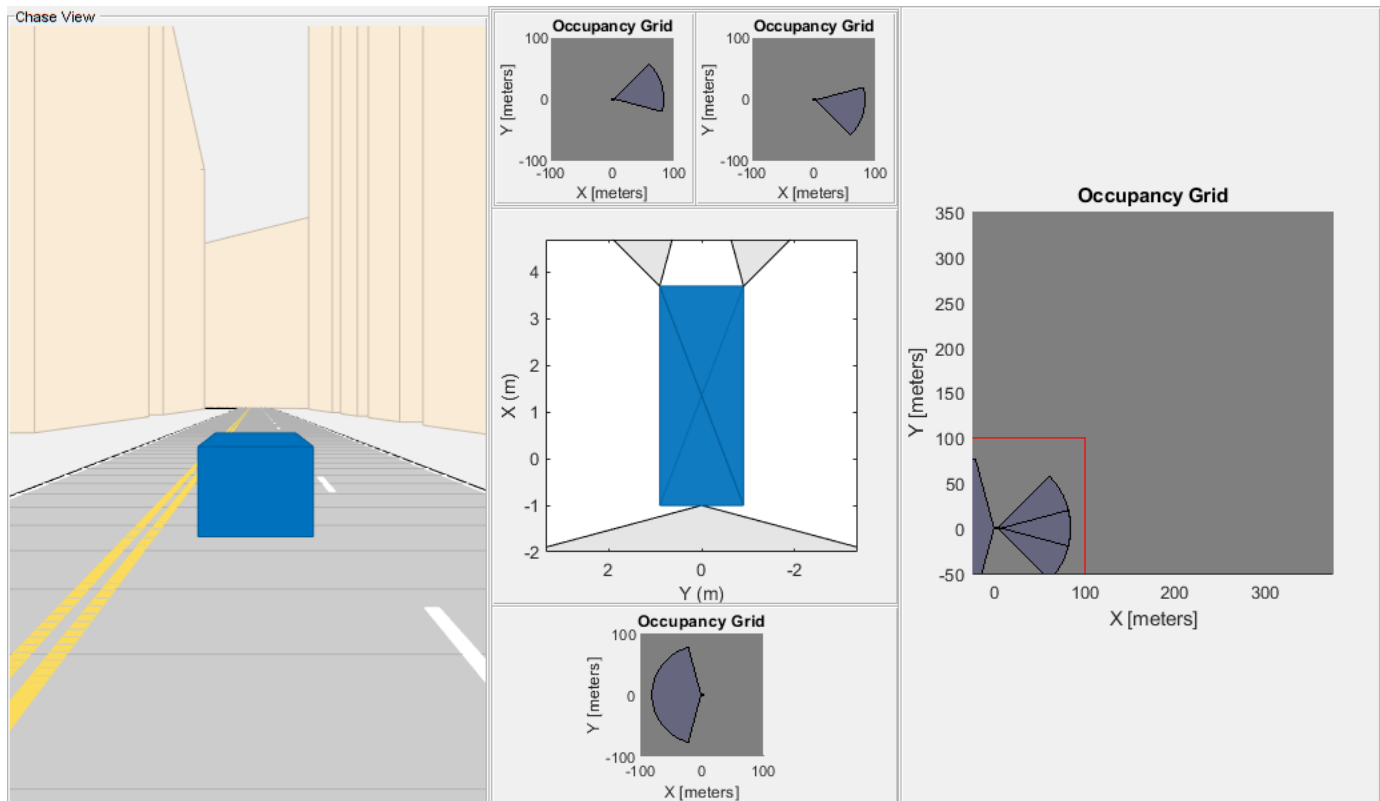


Simulate Sensor Readings and Build Map

Move the robot along the trajectory while updating the map with the simulated Lidar readings.

To run the driving scenario, call the `exampleHelperResetSimulation` helper function. This resets the simulation and trajectory, clears the map, and moves the egocentric maps back to the first point of the trajectory.

```
exampleHelperResetSimulation(scene, traj, lidarSensors, egoMap, reconLayer)
```

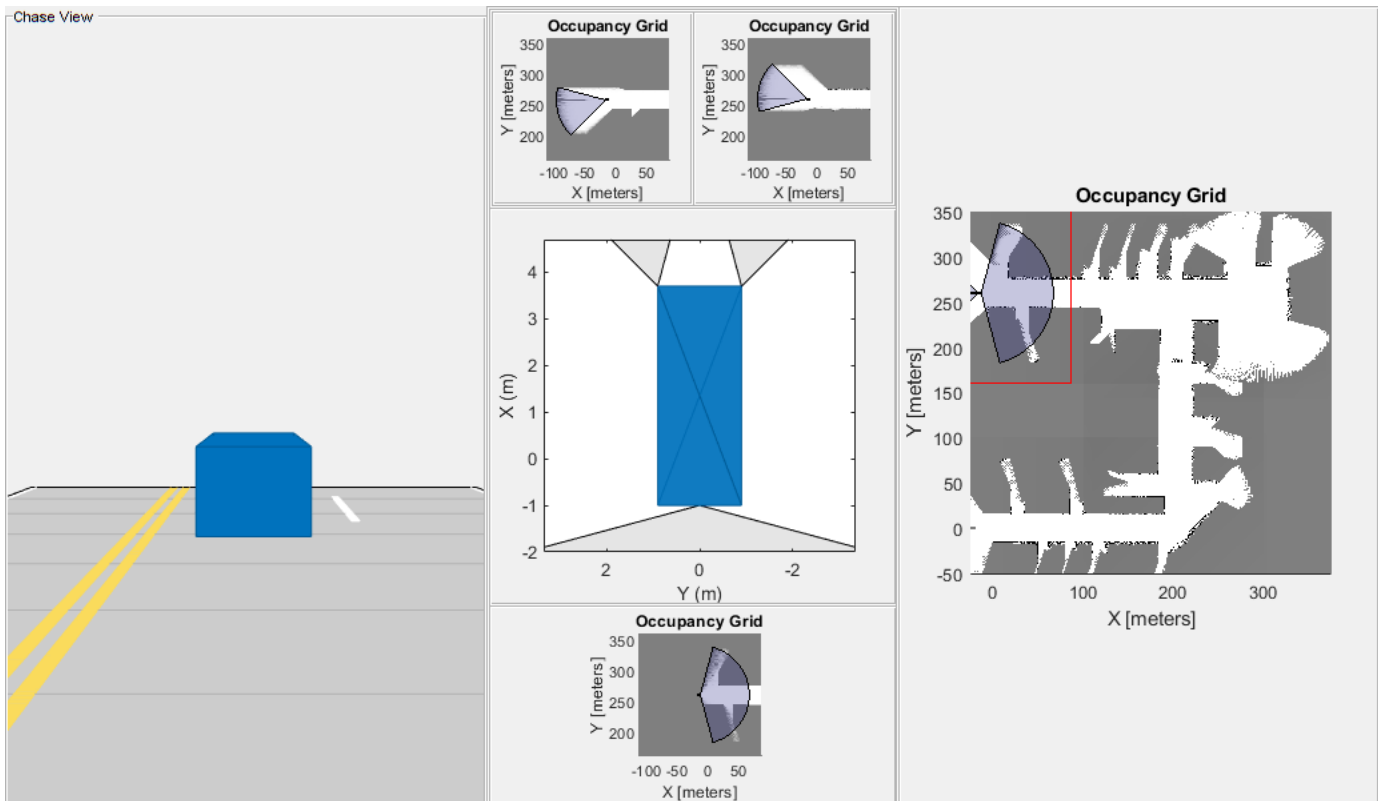


Call the `exampleHelperRunSimulation` function to execute the simulation.

The primary operations of the simulation loop are:

- Get the next pose in the trajectory from `traj` and extract the z-axis orientation (`theta`) from the quaternion.
- Move the `egoMap` to the new `[x y theta]` pose.
- Retrieve sensor data from the `lidarPointCloudGenerators`.
- Update the local maps with sensor data using `insertRay`.
- Update the global map using the `mapLayer` fused result.
- Refresh the visualization.

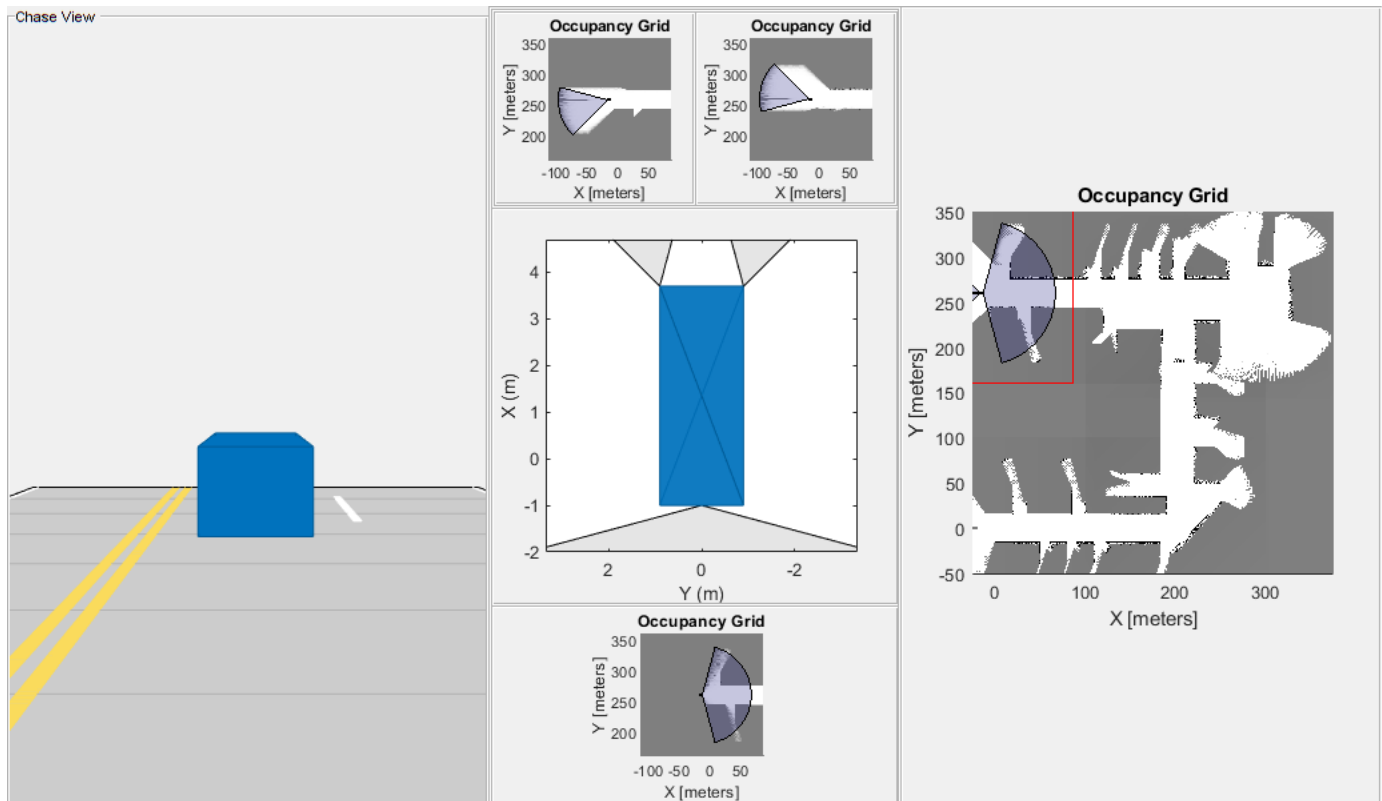
```
exampleHelperRunSimulation(scene, traj, groundTruthVehicle, egoMap, lidarSensors, reconLayer, axList)
```



The displayed results indicate that the front-right sensor is introducing large amounts of noise into the fused map. Notice that the right-hand wall has more variability throughout the trajectory. You do not want to discard readings from this sensor entirely because the sensor is still detecting free space in the front. Instead reduce the weight of those sensor readings during fusion and recreate the full multilayer map. Then, reset and rerun the simulation.

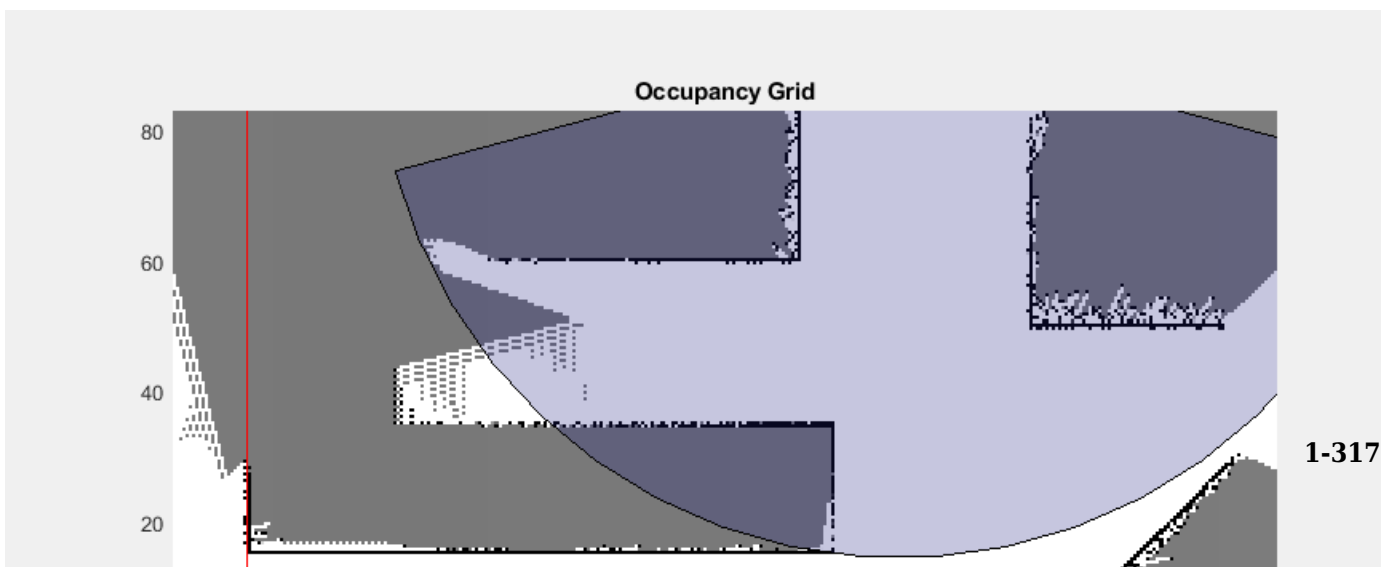
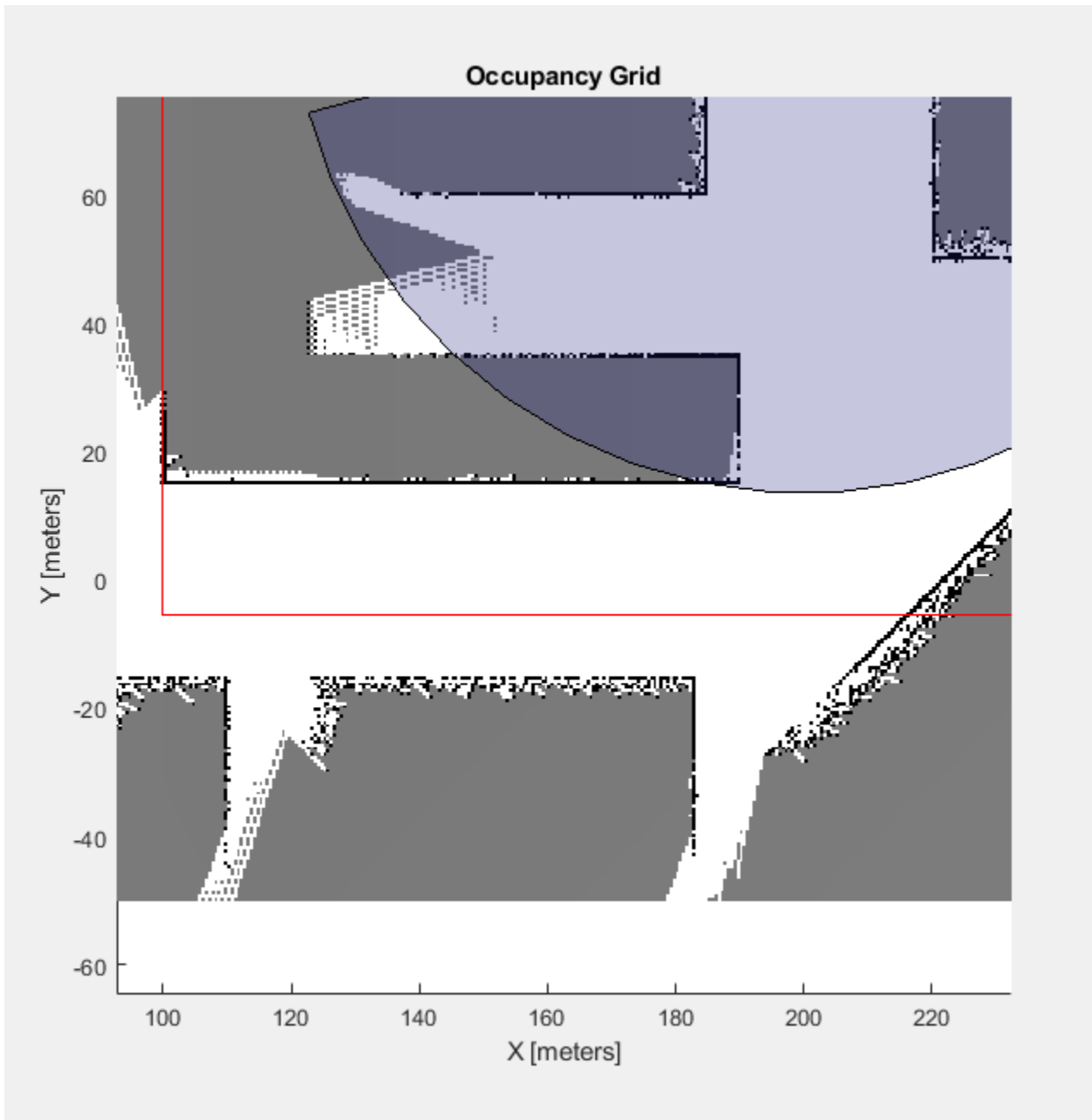
```
% Construct a new multiLayerMap with a different set of fusion weights
updatedWeights = [1 0.25 1];
egoMap = exampleHelperConstructMultiLayerEgoMap(res,width,height,updatedWeights);

% Rerun the simulation
exampleHelperResetSimulation(scene,traj,lidarSensors,egoMap,reconLayer)
exampleHelperRunSimulation(scene,traj,groundTruthVehicle,egoMap,lidarSensors,reconLayer,axList)
```



After simulating again, notice a few things about the map:

- Regions covered only by the noisy sensor can still detect freespace with little noise.
- While noise is still present, the readings from the other sensors outweigh those from the noisy sensor. The map shows distinct obstacle boundaries (black squares) in regions of sensor overlap.
- Noise beyond the distinct boundaries remain because the noisy lidar is the only sensor that reports readings in those areas, but does not connect to other free space.



Next Steps

This example shows a simple method of how readings can be fused. You may further customize this fusion with the following suggestions:

- To adjust weights based on sensor confidence prior to layer-layer fusion, specify a custom *inverse sensor model* when using the `insertRay` object function in the `exampleHelperUpdateEgoMap` function.
- To assign occupancy values based on a more complex confidence distribution like a gaussian inverse model, use the `raycast` object function to retrieve the cells traced by each emanating ray. Once a set of cells has been retrieved, values can be explicitly assigned based on more complex methods.
- To reduce confidence of aging cells, utilize additional map layers which keep track of timestamps for each cell. These timestamps can be used to place greater importance on recently updates cells and slowly ignore older readings.

Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans

This example demonstrates how to implement the Simultaneous Localization And Mapping (SLAM) algorithm on a collected series of lidar scans using pose graph optimization. The goal of this example is to build a map of the environment using the lidar scans and retrieve the trajectory of the robot.

To build the map of the environment, the SLAM algorithm incrementally processes the lidar scans and builds a pose graph that links these scans. The robot recognizes a previously-visited place through scan matching and may establish one or more loop closures along its moving path. The SLAM algorithm utilizes the loop closure information to update the map and adjust the estimated robot trajectory.

Load Laser Scan Data from File

Load a down-sampled data set consisting of laser scans collected from a mobile robot in an indoor environment. The average displacement between every two scans is around 0.6 meters.

The `offlineSlamData.mat` file contains the `scans` variable, which contains all the laser scans used in this example.

```
load('offlineSlamData.mat');
```

A floor plan and approximate path of the robot are provided for illustrative purposes. This image shows the relative environment being mapped and the approximate trajectory of the robot.



Run SLAM Algorithm, Construct Optimized Map and Plot Trajectory of the Robot

Create a `lidarSLAM` object and set the map resolution and the max lidar range. This example uses a Jackal™ robot from Clearpath Robotics™. The robot is equipped with a SICK™ TiM-511 laser scanner with a max range of 10 meters. Set the max lidar range slightly smaller than the max scan range (8m), as the laser readings are less accurate near max range. Set the grid map resolution to 20 cells per meter, which gives a 5cm precision.

```
maxLidarRange = 8;  
mapResolution = 20;  
slamAlg = lidarSLAM(mapResolution, maxLidarRange);
```

The following loop closure parameters are set empirically. Using higher loop closure threshold helps reject false positives in loop closure identification process. However, keep in mind that a high-score match may still be a bad match. For example, scans collected in an environment that has similar or repeated features are more likely to produce false positives. Using a higher loop closure search radius allows the algorithm to search a wider range of the map around current pose estimate for loop closures.

```
slamAlg.LoopClosureThreshold = 210;  
slamAlg.LoopClosureSearchRadius = 8;
```

Observe the Map Building Process with Initial 10 Scans

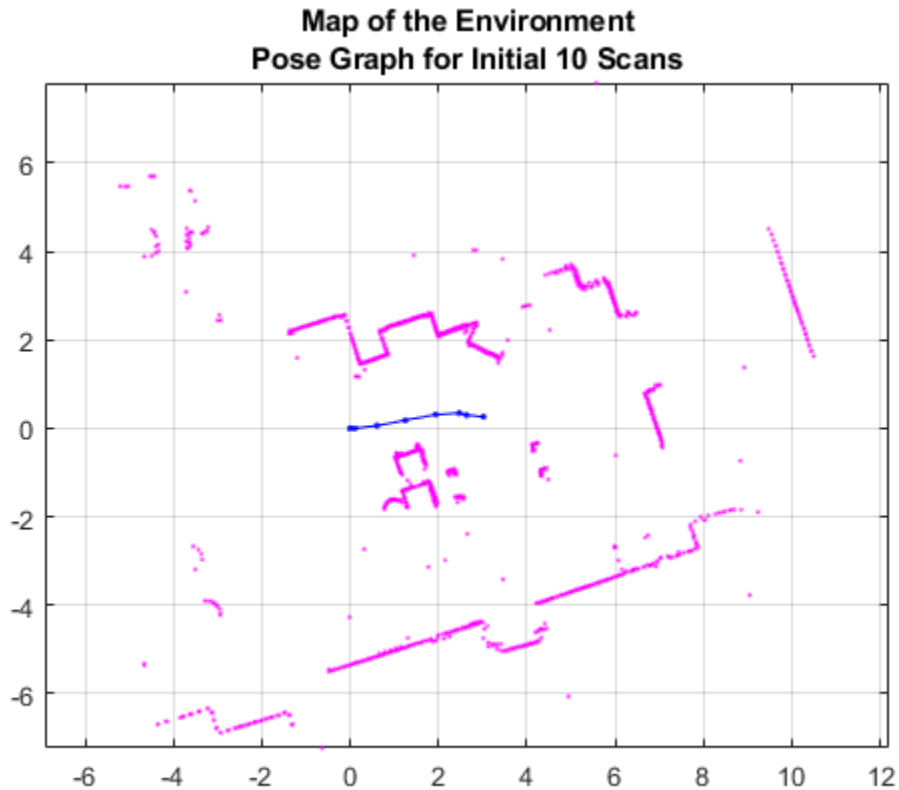
Incrementally add scans to the `slamAlg` object. Scan numbers are printed if added to the map. The object rejects scans if the distance between scans is too small. Add the first 10 scans first to test your algorithm.

```
for i=1:10  
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});  
    if isScanAccepted  
        fprintf('Added scan %d \n', i);  
    end  
end
```

```
Added scan 1  
Added scan 2  
Added scan 3  
Added scan 4  
Added scan 5  
Added scan 6  
Added scan 7  
Added scan 8  
Added scan 9  
Added scan 10
```

Reconstruct the scene by plotting the scans and poses tracked by the `slamAlg`.

```
figure;  
show(slamAlg);  
title({'Map of the Environment', 'Pose Graph for Initial 10 Scans'});
```

Observe the Effect of Loop Closures and the Optimization Process

Continue to add scans in a loop. Loop closures should be automatically detected as the robot moves. Pose graph optimization is performed whenever a loop closure is identified. The output `optimizationInfo` has a field, `IsPerformed`, that indicates when pose graph optimization occurs.

Plot the scans and poses whenever a loop closure is identified and verify the results visually. This plot shows overlaid scans and an optimized pose graph for the first loop closure. A loop closure edge is added as a red link.

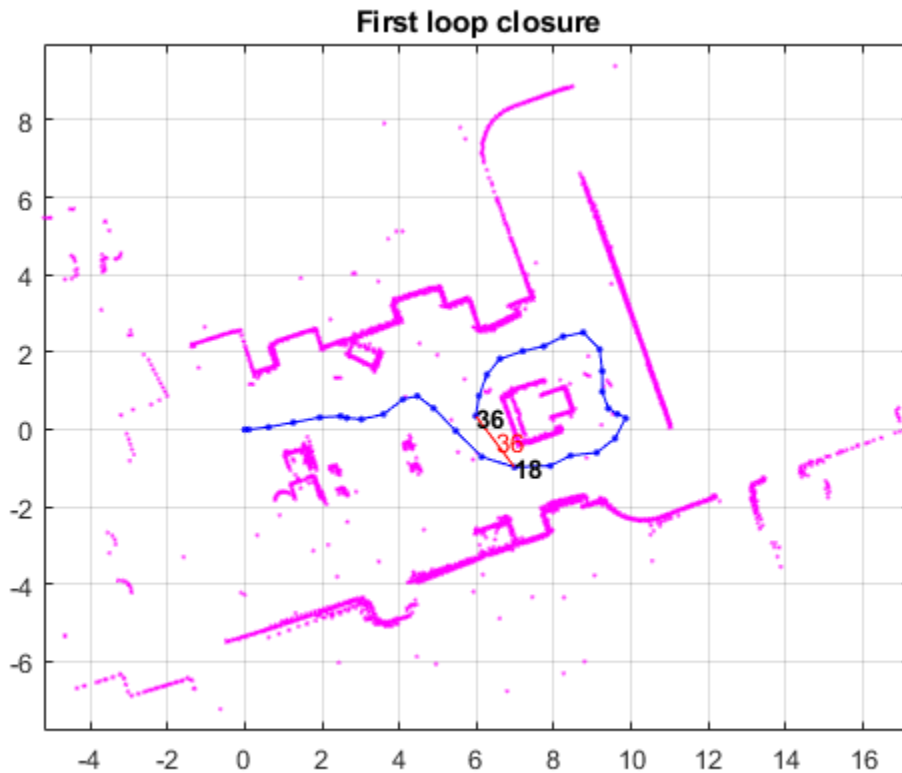
```

firstTimeLCDetected = false;

figure;
for i=10:length(scans)
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});
    if ~isScanAccepted
        continue;
    end
    % visualize the first detected loop closure, if you want to see the
    % complete map building process, remove the if condition below
    if optimizationInfo.IsPerformed && ~firstTimeLCDetected
        show(slamAlg, 'Poses', 'off');
        hold on;
        show(slamAlg.PoseGraph);
        hold off;
        firstTimeLCDetected = true;
        drawnow
    end
end

```

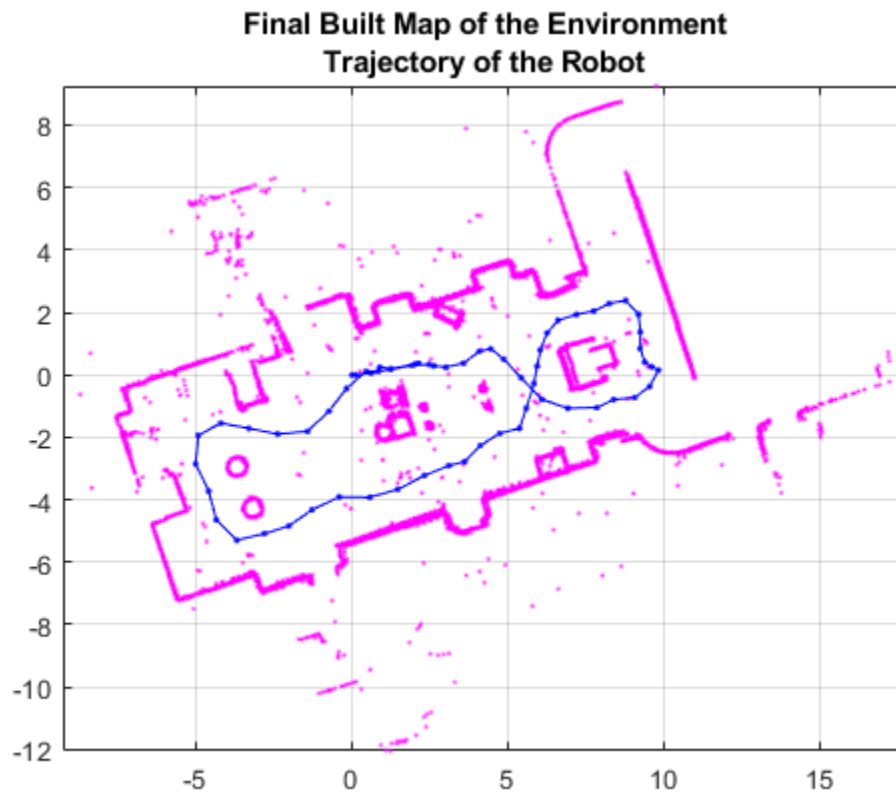
```
end  
end  
title('First loop closure');
```



Visualize the Constructed Map and Trajectory of the Robot

Plot the final built map after all scans are added to the `slamAlg` object. Though the previous `for` loop only plotted the initial closure, all the scans were added.

```
figure  
show(slamAlg);  
title({'Final Built Map of the Environment', 'Trajectory of the Robot'});
```



Visually Inspect the Built Map Compared to the Original Floor Plan

An image of the scans and pose graph is overlaid on the original floorplan. You can see that the map matches the original floor plan well after adding all the scans and optimizing the pose graph.



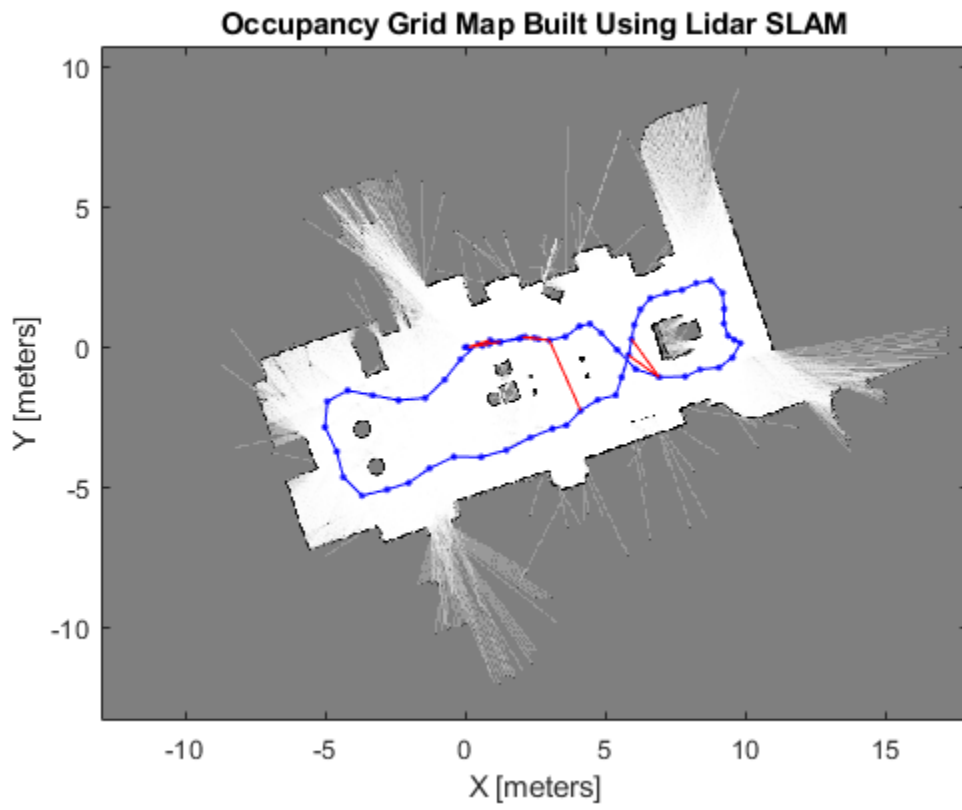
Build Occupancy Grid Map

The optimized scans and poses can be used to generate a `occupancyMap`, which represents the environment as a probabilistic occupancy grid.

```
[scans, optimizedPoses] = scansAndPoses(slamAlg);  
map = buildMap(scans, optimizedPoses, mapResolution, maxLidarRange);
```

Visualize the occupancy grid map populated with the laser scans and the optimized pose graph.

```
figure;  
show(map);  
hold on  
show(slamAlg.PoseGraph, 'IDs', 'off');  
hold off  
title('Occupancy Grid Map Built Using Lidar SLAM');
```



Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans

This example demonstrates how to implement the Simultaneous Localization And Mapping (SLAM) algorithm on lidar scans obtained from simulated environment using pose graph optimization. This example requires Simulink® 3D Animation™ and Navigation Toolbox™.

The goal of this example is to build a map of the environment using the lidar scans and retrieve the trajectory of the robot, with the robot simulator in the loop.

The basics of SLAM algorithm can be found in the “Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans” on page 1-319 example.

Load Trajectory of the Robot from File

The robot trajectory are waypoints given to the robot to move in the simulated environment. For this example, the robot trajectory is provided for you.

```
load slamRobotTrajectory.mat
```

A floor plan and approximate path of the robot are provided for illustrative purposes. This image shows the environment being mapped and the approximate trajectory of the robot.

Load and View the Virtual World

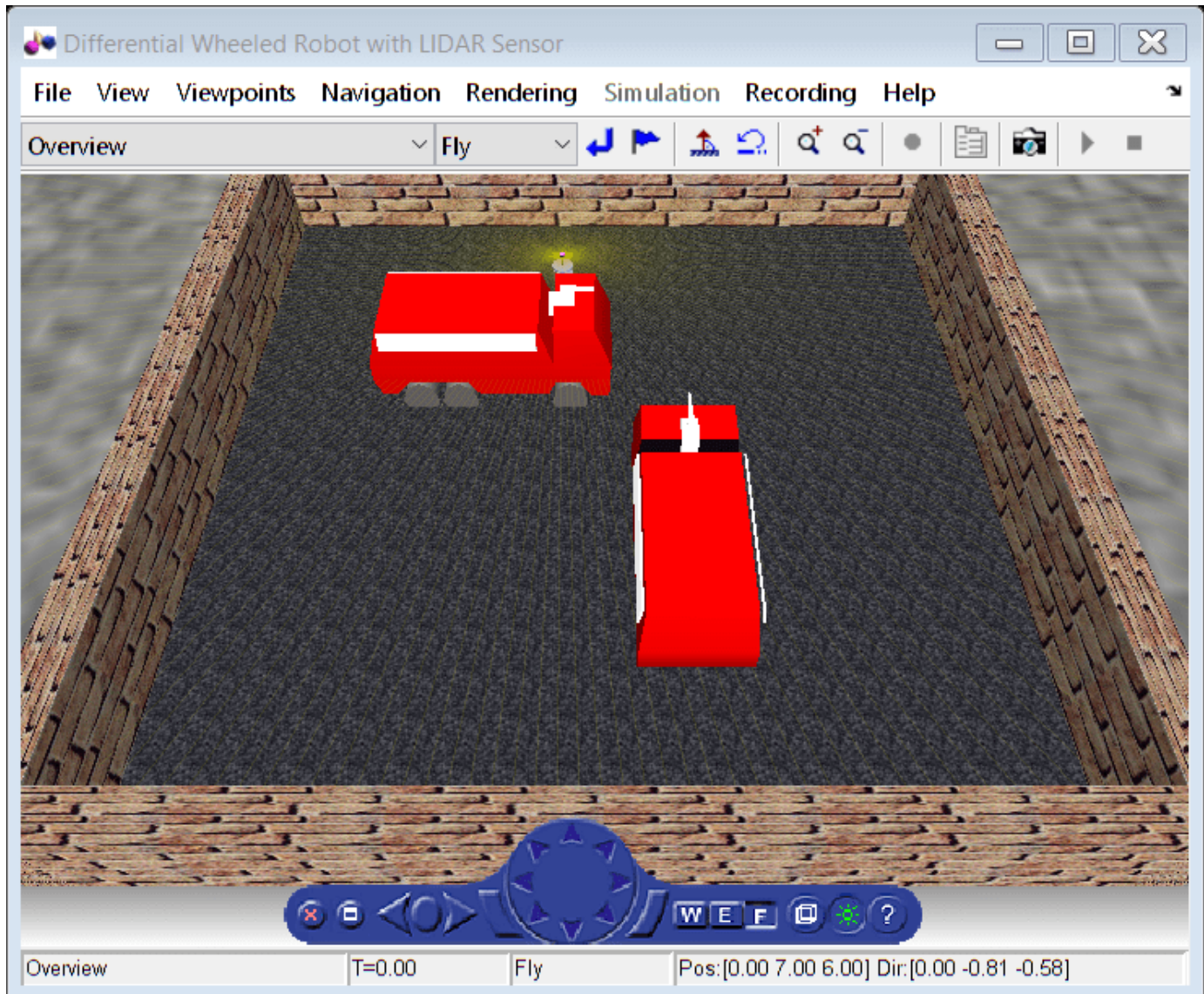
This example uses a virtual scene with two vehicles and four walls as obstacles and a robot equipped with a lidar scanner shown in the Simulink 3D Animation Viewer. You can navigate in a virtual scene using the menu bar, toolbar, navigation panel, mouse, and keyboard. Key features of the viewer are illustrated in the “Plane Manipulation Using Space Mouse MATLAB Object” (Simulink 3D Animation) example.

Create and open the vrworld object.

```
w = vrworld('slamSimulatedWorld.x3d');  
open(w)
```

Create a figure showing the virtual scene

```
vrf = vrfigure(w)
```



vrf =

vrfigure object: 1-by-1

Differential Wheeled Robot with LIDAR Sensor

Initialize the Robot Position and Rotation in Virtual World

The virtual scene is represented as the hierarchical structure of a VRML file used by Simulink 3D Animation. The position and orientation of child objects is relative to the parent object. The robot `vrnode` is used to manipulate the position and orientation of the robot in the virtual scene.

To access a VRML node, an appropriate `vrnode` object must be created. The node is identified by its name and the world it belongs to.

Create `vrnode` handle for robot in virtual environment.

```
robotVRNode = vrnnode(w, 'Robot');
```

Set the initial position of the robot from the trajectory first point and set the initial rotation to 0 rad about y axis.

```
robotVRNode.children.translation = [trajectory(1,1) 0 trajectory(1,2)];  
robotVRNode.children.rotation = [0 1 0 0];
```

Create handle for lidar sensor on robot by creating vrnnode.

```
lidarVRNode = vrnnode(w, 'LIDAR_Sensor');
```

The simulated lidar is using total 240 laser lines and the angle between these lines is 1.5 degree.

```
angles = 180:-1.5:-178.5;  
angles = deg2rad(angles)';
```

Waiting to update and initialize virtual scene

```
pause(1)
```

Create Lidar Slam Object

Create a `lidarSLAM` object and set the map resolution and the max lidar range. This example uses a simulated virtual environment. The robot in this `vrworld` has a lidar sensor with range of 0 to 10 meters. Set the max lidar range (8m) smaller than the max scan range, as the laser readings are less accurate near max range. Set the grid map resolution to 20 cells per meter, which gives a 5cm precision. These two parameters are used throughout the example.

```
maxLidarRange = 8;  
mapResolution = 20;  
slamAlg = lidarSLAM(mapResolution,maxLidarRange);
```

The loop closure parameters are set empirically. Using a higher loop closure threshold helps reject false positives in loop closure identification process. Keep in mind that a high-score match may still be a bad match. For example, scans collected in an environment that has similar or repeated features are more likely to produce false positive. Using a higher loop closure search radius allows the algorithm to search a wider range of the map around the current pose estimate for loop closures.

```
slamAlg.LoopClosureThreshold = 200;  
slamAlg.LoopClosureSearchRadius = 3;  
controlRate = rateControl(10);
```

Observe the Effect of Loop Closure and Optimization Process

Create a loop to navigate the robot through the virtual scene. The robot position is updated in the loop from the trajectory points. The scans are obtained from the robot as robot navigates through the environment.

Loop closures are automatically detected as the robot moves. The pose graph optimization is performed whenever a loop closure is detected. This can be checked using the output `optimizationInfo.IsPerformed` value from `addScan`.

A snapshot is shown to demonstrate of the scans and poses when the first loop closure is identified and verify the results visually. This plot shows overlaid scans and an optimized pose graph for the first loop closure.

The final built map would be presented after all the scans are collected and processed.

The plot is updated continuously as robot navigates through virtual scene

```

firstLoopClosure = false;
scans = cell(length(trajectory),1);

figure
for i=1:length(trajectory)
    % Use translation property to move the robot.
    robotVRNode.children.translation = [trajectory(i,1) 0 trajectory(i,2)];
    vrdrawnow;

    % Read the range readings obtained from lidar sensor of the robot.
    range = lidarVRNode.pickedRange;

    % The simulated lidar readings will give -1 values if the objects are
    % out of range. Make all these value to the greater than
    % maxLidarRange.
    range(range==-1) = maxLidarRange+2;

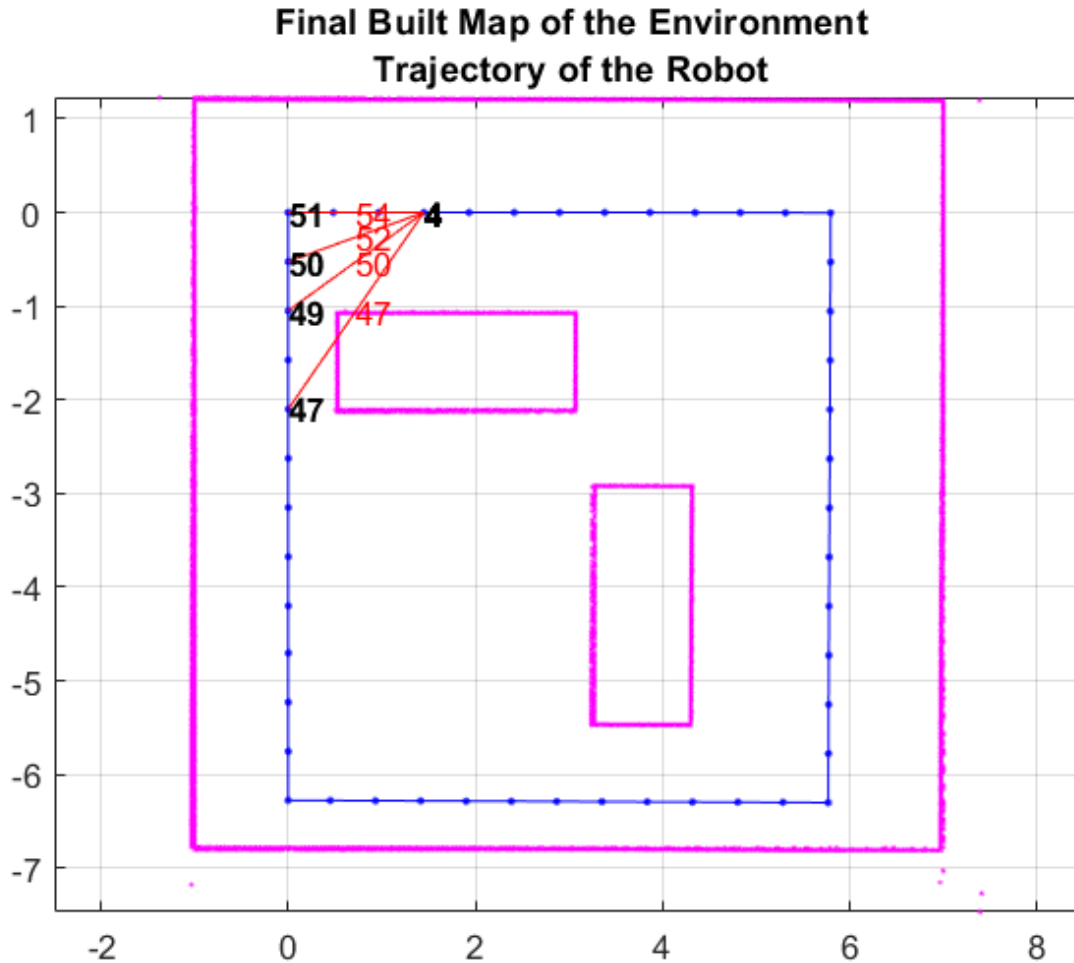
    % Create a lidarScan object from the ranges and angles.
    scans{i} = lidarScan(range,angles);

    [isScanAccepted,loopClosureInfo,optimizationInfo] = addScan(slamAlg,scans{i});
    if isScanAccepted
        % Visualize how scans plot and poses are updated as robot navigates
        % through virtual scene
        show(slamAlg);

        % Visualize the first detected loop closure
        % firstLoopClosure flag is used to capture the first loop closure event
        if optimizationInfo.IsPerformed && ~firstLoopClosure
            firstLoopClosure = true;
            show(slamAlg,'Poses','off');
            hold on;
            show(slamAlg.PoseGraph);
            hold off;
            title('First loop closure');
            snapnow
        end
    end

    waitfor(controlRate);
end

```

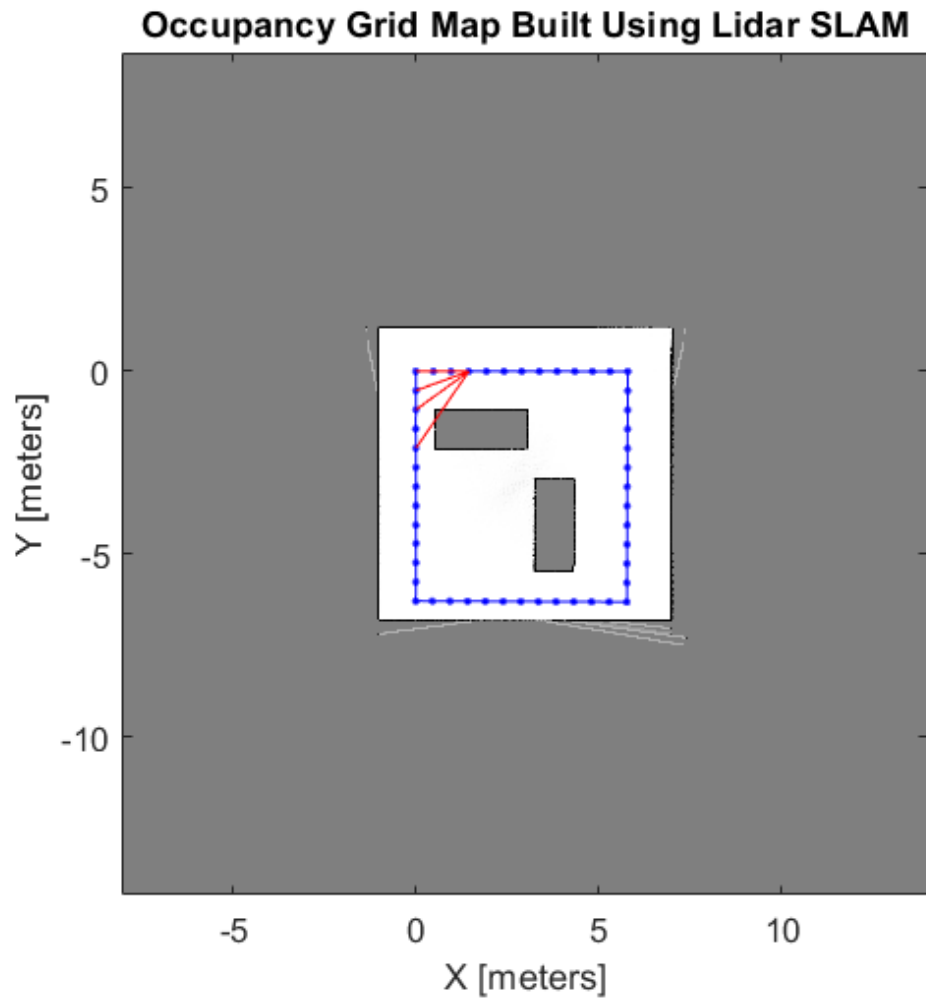
Build Occupancy Grid Map

The optimized scans and poses can be used to generate a `occupancyMap` which represents the environment as a probabilistic occupancy grid.

```
[scans,optimizedPoses] = scansAndPoses(slamAlg);
map = buildMap(scans,optimizedPoses,mapResolution,maxLidarRange);
```

Visualize the occupancy grid map populated with the laser scans and the optimized pose graph.

```
figure;
show(map);
hold on
show(slamAlg.PoseGraph,'IDs','off');
hold off
title('Occupancy Grid Map Built Using Lidar SLAM');
```



Close the virtual scene.

```
close(vrf);  
close(w);  
delete(w);
```

Perform SLAM Using 3-D Lidar Point Clouds

This example demonstrates how to implement the *simultaneous localization and mapping* (SLAM) algorithm on collected 3-D lidar sensor data using point cloud processing algorithms and pose graph optimization. The goal of this example is to estimate the trajectory of the robot and create a 3-D occupancy map of the environment from the 3-D lidar point clouds and estimated trajectory.

The demonstrated SLAM algorithm estimates a trajectory using a Normal Distribution Transform (NDT) based point cloud registration algorithm and reduces the drift using SE3 pose graph optimization using trust-region solver whenever a robot revisits a place.

Load Data And Set Up Tunable Parameters

Load the 3-D lidar data collected from a Clearpath™ Husky robot in a parking garage. The lidar data contains a cell array of n -by-3 matrices, where n is the number 3-D points in the captured lidar data, and the columns represent xyz-coordinates associated with each captured point.

```
load pClouds.mat
```

Parameters For Point Cloud Registration Algorithm

Specify the parameters for estimating the trajectory using the point cloud registration algorithm. `maxLidarRange` specifies the maximum range of the 3-D laser scanner.

```
maxLidarRange = 20;
```

The point cloud data captured in an indoor environment contains points lying on the ground and ceiling planes, which confuses the point cloud registration algorithms. Some points are removed from the point cloud with these parameters:

- `referenceVector` - Normal to the ground plane.
- `maxDistance` - Maximum distance for inliers when removing the ground and ceiling planes.
- `maxAngularDistance` - Maximum angle deviation from the reference vector when fitting the ground and ceiling planes.

```
referenceVector = [0 0 1];
maxDistance = 0.5;
maxAngularDistance = 15;
```

To improve the efficiency and accuracy of the registration algorithm, the point clouds are downsampled using random sampling with a sample ratio specified by `randomSampleRatio`.

```
randomSampleRatio = 0.25;
```

`gridStep` specifies the voxel grid sizes used in the NDT registration algorithm. A scan is accepted only after the robot moves by a distance greater than `distanceMovedThreshold`.

```
gridStep = 2.5;
distanceMovedThreshold = 0.3;
```

Tune these parameters for your specific robot and environment.

Parameters For Loop Closure Estimation Algorithm

Specify the parameters for the loop closure estimation algorithm. Loop closures are only searched within a radius around the current robot location specified by `loopClosureSearchRadius`.

```
loopClosureSearchRadius = 3;
```

The loop closure algorithm is based on 2-D submap and scan matching. A submap is created after every `nScansPerSubmap` (Number of Scans per submap) accepted scans. The loop closure algorithm disregards the most recent `subMapThresh` scans while searching for loop candidates.

```
nScansPerSubmap = 3;  
subMapThresh = 50;
```

An annular region with z-limits specified by `annularRegionLimits` is extracted from the point clouds. Points outside these limits on the floor and ceiling are removed after the point cloud plane fit algorithms identify the region of interest.

```
annularRegionLimits = [-0.75 0.75];
```

The maximum acceptable Root Mean Squared Error (RMSE) in estimating relative pose between loop candidates is specified by `rmseThreshold`. Choose a lower value for estimating accurate loop closure edges, which has a high impact on pose graph optimization.

```
rmseThreshold = 0.26;
```

The threshold over scan matching score to accept a loop closure is specified by `loopClosureThreshold`. Pose Graph Optimization is called after inserting `optimizationInterval` loop closure edges into the pose graph.

```
loopClosureThreshold = 150;  
optimizationInterval = 2;
```

Initialize Variables

Set up a pose graph, occupancy map, and necessary variables.

```
% 3D Posegraph object for storing estimated relative poses  
pGraph = poseGraph3D;  
% Default serialized upper-right triangle of 6-by-6 Information Matrix  
infoMat = [1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,1,0,0,1];  
% Number of loop closure edges added since last pose graph optimization and map refinement  
numLoopClosuresSinceLastOptimization = 0;  
% True after pose graph optimization until the next scan  
mapUpdated = false;  
% Equals to 1 if the scan is accepted  
scanAccepted = 0;  
  
% 3D Occupancy grid object for creating and visualizing 3D map  
mapResolution = 8; % cells per meter  
omap = occupancyMap3D(mapResolution);
```

Preallocate variables for the processed point clouds, lidar scans, and submaps. Create a downsampled set of point clouds for quickly visualizing the map.

```
pcProcessed = cell(1,length(pClouds));  
lidarScans2d = cell(1,length(pClouds));  
submaps = cell(1,length(pClouds)/nScansPerSubmap);  
  
pcsToView = cell(1,length(pClouds));
```

Create variables for display purposes.

```

% Set to 1 to visualize created map and posegraph during build process
viewMap = 1;
% Set to 1 to visualize processed point clouds during build process
viewPC = 0;

Set random seed to guarantee consistent random sampling.

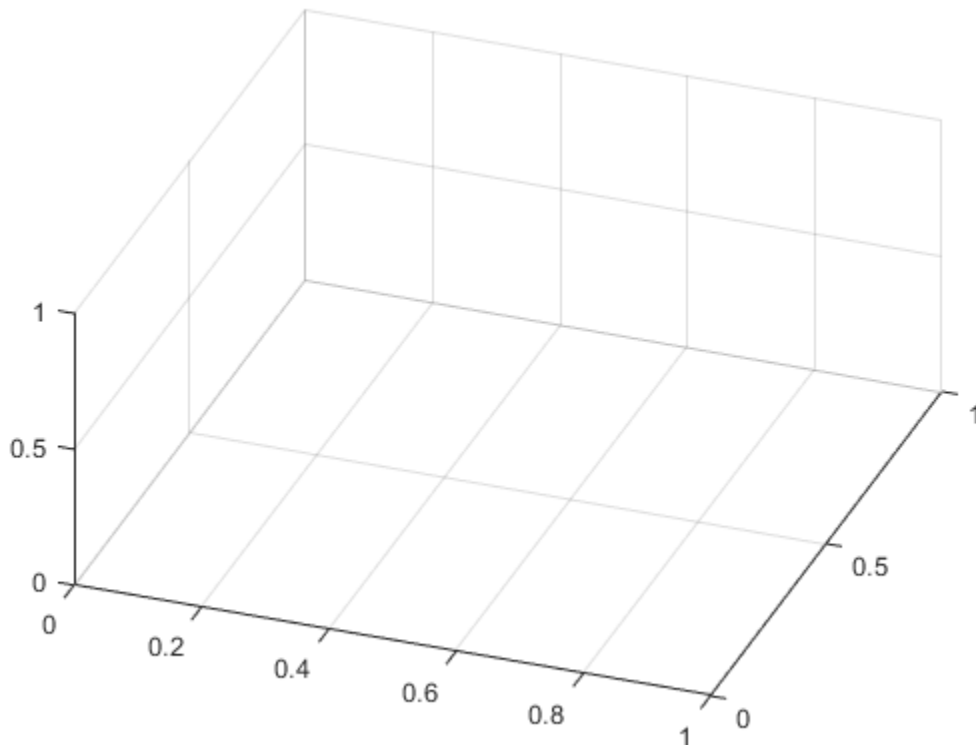
rng(0);

Initialize figure windows if desired.

% If you want to view the point clouds while processing them sequentially
if viewPC==1
    pplayer = pcplayer([-50 50],[-50 50],[-10 10], 'MarkerSize',10);
end

% If you want to view the created map and posegraph during build process
if viewMap==1
    ax = newplot; % Figure axis handle
    view(20,50);
    grid on;
end

```



Trajectory Estimation And Refinement Using Pose Graph Optimization

The trajectory of the robot is a collection of robot poses (location and orientation in 3-D space). A robot pose is estimated at every 3-D lidar scan acquisition instance using the 3-D lidar SLAM algorithm. The 3-D lidar SLAM algorithm has the following steps:

- Point cloud filtering
- Point cloud downsampling
- Point cloud registration
- Loop closure query
- Pose graph optimization

Iteratively process the point clouds to estimate the trajectory.

```
count = 0; % Counter to track number of scans added
disp('Estimating robot trajectory...');
```

```
Estimating robot trajectory...
```

```
for i=1:length(pClouds)
    % Read point clouds in sequence
    pc = pClouds{i};
```

Point Cloud Filtering

Point cloud filtering is done to extract the region of interest from the acquired scan. In this example, the region of interest is the annular region with ground and ceiling removed.

Remove invalid points outside the max range and unnecessary points behind the robot corresponding to the human driver.

```
ind = (-maxLidarRange < pc(:,1) & pc(:,1) < maxLidarRange ...
    & -maxLidarRange < pc(:,2) & pc(:,2) < maxLidarRange ...
    & (abs(pc(:,2))>abs(0.5*pc(:,1)) | pc(:,1)>0));

pcl = pointCloud(pc(ind,:));
```

Remove points on the ground plane.

```
[~, ~, outliers] = ...
    pcfplane(pcl, maxDistance, referenceVector, maxAngularDistance);
pcl_wogr = select(pcl, outliers, 'OutputSize', 'full');
```

Remove points on the ceiling plane.

```
[~, ~, outliers] = ...
    pcfplane(pcl_wogr, 0.2, referenceVector, maxAngularDistance);
pcl_wogr = select(pcl_wogr, outliers, 'OutputSize', 'full');
```

Select points in annular region.

```
ind = (pcl_wogr.Location(:,3)<annularRegionLimits(2))&(pcl_wogr.Location(:,3)>annularRegion
pcl_wogr = select(pcl_wogr, ind, 'OutputSize', 'full');
```

Point Cloud Downsampling

Point cloud downsampling improves the speed and accuracy of the point cloud registration algorithm. Down sampling should be tuned for specific needs. The random sampling algorithm is chosen empirically from down sampling variants below for the current scenario.

```
pcl_wogr_sampled = pcdsample(pcl_wogr, 'random', randomSampleRatio);
```



```

if viewPC==1
    % Visualize down sampled point cloud
    view(pplayer,pcl_wogrd_sampled);
    pause(0.001)
end

```

Point Cloud Registration

Point cloud registration estimates the relative pose (rotation and translation) between current scan and previous scan. The first scan is always accepted (processed further and stored) but the other scans are only accepted after translating more than the specified threshold. `poseGraph3D` is used to store the estimated accepted relative poses (trajectory).

```

if count == 0
    % First scan
    tform = [];
    scanAccepted = 1;
else
    if count == 1
        tform = pcregisterndt(pcl_wogrd_sampled,prevPc,gridStep);
    else
        tform = pcregisterndt(pcl_wogrd_sampled,prevPc,gridStep,...
            'InitialTransform',prevTform);
    end
    relPose = [tform2trvec(tform.T') tform2quat(tform.T')];

    if sqrt(norm(relPose(1:3))) > distanceMovedThreshold
        addRelativePose(pGraph,relPose);
        scanAccepted = 1;
    else
        scanAccepted = 0;
    end
end
end

```

Loop Closure Query

Loop closure query determines whether or not the current robot location has previously been visited. The search is performed by matching the current scan with the previous scans within a small radius around the current robot location specified by `loopClosureSearchRadius`. Searching within the small radius is sufficient because of the low-drift in lidar odometry, as searching against all previous scans is time consuming. Loop closure query consists of the following steps:

- Create submaps from `nScansPerSubmap` consecutive scans.
- Match the current scan with the submaps within the `loopClosureSearchRadius`.
- Accept the matches if the match score is greater than the `loopClosureThreshold`. All the scans representing accepted submap are considered as probable loop candidates.
- Estimate the relative pose between probable loop candidates and the current scan. A relative pose is accepted as a loop closure constraint only when the RMSE is less than the `rmseThreshold`.

```

if scanAccepted == 1
    count = count + 1;

    pcProcessed{count} = pcl_wogrd_sampled;

```

```

lidarScans2d{count} = exampleHelperCreate2DScan(pcl_wogrd_sampled);

% Submaps are created for faster loop closure query.
if rem(count,nScansPerSubmap)==0
    submaps{count/nScansPerSubmap} = exampleHelperCreateSubmap(lidarScans2d,...
        pGraph,count,nScansPerSubmap,maxLidarRange);
end

% loopSubmapIds contains matching submap ids if any otherwise empty.
if (floor(count/nScansPerSubmap)>subMapThresh)
    [loopSubmapIds,~] = exampleHelperEstimateLoopCandidates(pGraph,...
        count,submaps,lidarScans2d{count},nScansPerSubmap,...
        loopClosureSearchRadius,loopClosureThreshold,subMapThresh);

    if ~isempty(loopSubmapIds)
        rmseMin = inf;

        % Estimate best match to the current scan
        for k = 1:length(loopSubmapIds)
            % For every scan within the submap
            for j = 1:nScansPerSubmap
                probableLoopCandidate = ...
                    loopSubmapIds(k)*nScansPerSubmap - j + 1;
                [loopTform,~,rmse] = pregisterndt(pcl_wogrd_sampled,...
                    pcProcessed{probableLoopCandidate},gridStep);
                % Update best Loop Closure Candidate
                if rmse < rmseMin
                    loopCandidate = probableLoopCandidate;
                    rmseMin = rmse;
                end
            end
            if rmseMin < rmseThreshold
                break;
            end
        end
        end
        end

        % Check if loop candidate is valid
        if rmseMin < rmseThreshold
            % loop closure constraint
            relPose = [tform2trvec(loopTform.T') tform2quat(loopTform.T')];

            addRelativePose(pGraph,relPose,infoMat,...
                loopCandidate,count);
            numLoopClosuresSinceLastOptimization = numLoopClosuresSinceLastOptimization + 1;
        end
    end
end
end
end

```

Pose Graph Optimization

Pose graph optimization runs after a sufficient number of loop edges are accepted to reduce the drift in trajectory estimation. After every loop closure optimization the loop closure search radius is reduced due to the fact that the uncertainty in the pose estimation reduces after optimization.

```

if (numLoopClosuresSinceLastOptimization == optimizationInterval)||...
    ((numLoopClosuresSinceLastOptimization>0)&&(i==length(pClouds)))
    if loopClosureSearchRadius ~=1
        disp('Doing Pose Graph Optimization to reduce drift.');
```

end

```

% pose graph optimization
pGraph = optimizePoseGraph(pGraph);
loopClosureSearchRadius = 1;
if viewMap == 1
    position = pGraph.nodes;
    % Rebuild map after pose graph optimization
    omap = occupancyMap3D(mapResolution);
    for n = 1:(pGraph.NumNodes-1)
        insertPointCloud(omap,position(n,:),pcsToView{n}.removeInvalidPoints,maxLidarRange);
    end
    mapUpdated = true;
    ax = newplot;
    grid on;
end
numLoopClosuresSinceLastOptimization = 0;
% Reduce the frequency of optimization after optimizing
% the trajectory
optimizationInterval = optimizationInterval*7;
end
```

Visualize the map and pose graph during the build process. This visualization is costly, so enable it only when necessary by setting `viewMap` to 1. If visualization is enabled then the plot is updated after every 15 added scans.

```

pcToView = pcdsample(pcl_wogrd_sampled, 'random', 0.5);
pcsToView{count} = pcToView;

if viewMap==1
    % Insert point cloud to the occurance map in the right position
    position = pGraph.nodes(count);
    insertPointCloud(omap,position,pcToView.removeInvalidPoints,maxLidarRange);

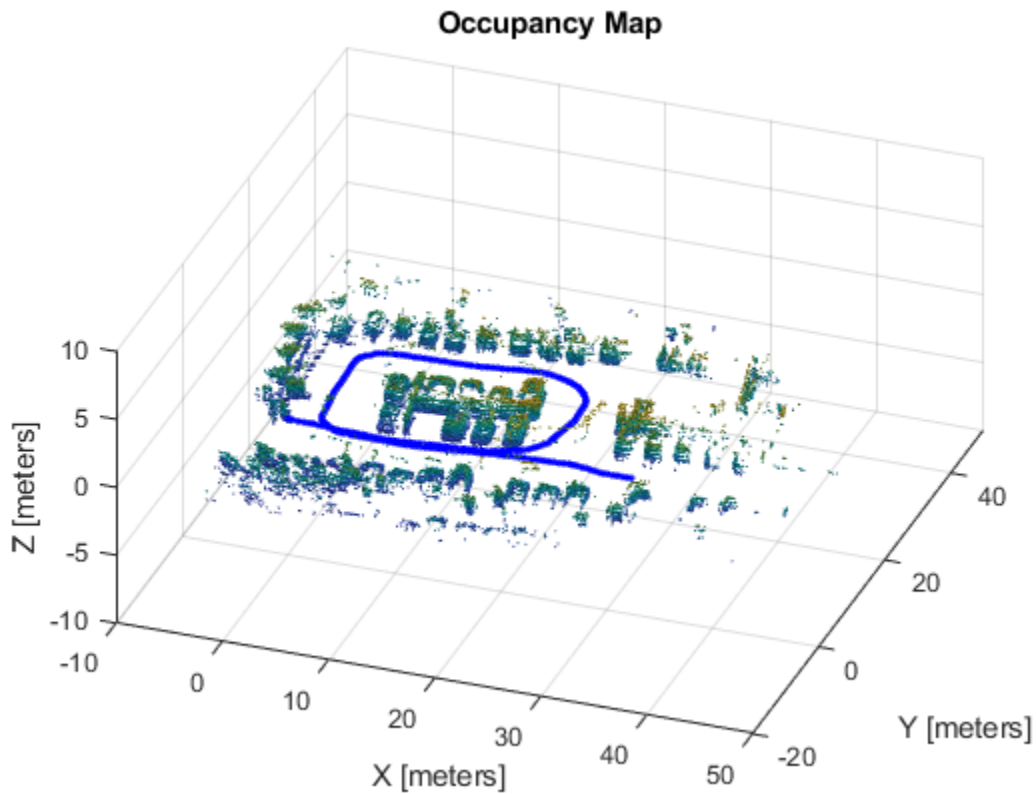
    if (rem(count-1,15)==0)||mapUpdated
        exampleHelperVisualizeMapAndPoseGraph(omap, pGraph, ax);
    end
    mapUpdated = false;
else
    % Give feedback to know that example is running
    if (rem(count-1,15)==0)
        fprintf('.');
    end
end
end
```

Update previous relative pose estimate and point cloud.

```

    prevPc = pcl_wogrd_sampled;
    prevTform = tform;
end
end
```

Doing Pose Graph Optimization to reduce drift.



Build and Visualize 3-D Occupancy Map

The point clouds are inserted into `occupancyMap3D` using the estimated global poses. After iterating through all the nodes, the full map and estimated vehicle trajectory is shown.

```

if (viewMap ~= 1) || (numLoopClosuresSinceLastOptimization > 0)
    nodesPositions = nodes(pGraph);
    % Create 3D Occupancy grid
    omapToView = occupancyMap3D(mapResolution);

    for i = 1:(size(nodesPositions,1)-1)
        pc = pcsToView{i};
        position = nodesPositions(i,:);

        % Insert point cloud to the occupance map in the right position
        insertPointCloud(omapToView, position, pc.removeInvalidPoints, maxLidarRange);
    end

    figure;
    axisFinal = newplot;
    exampleHelperVisualizeMapAndPoseGraph(omapToView, pGraph, axisFinal);
end

```

Plan Mobile Robot Paths Using RRT

This example shows how to use the rapidly exploring random tree (RRT) algorithm to plan a path for a vehicle through a known map. Special vehicle constraints are also applied with a custom state space. You can tune your own planner with custom state space and path validation objects for any navigation application.

Load Occupancy Map

Load an existing occupancy map of a small office space. Plot the start and goal poses of the vehicle on top of the map.

```
load("office_area_gridmap.mat","occGrid")
show(occGrid)

% Set start and goal poses.
start = [-1.0,0.0,-pi];
goal = [14,-2.25,0];

% Show start and goal positions of robot.
hold on
plot(start(1),start(2),'ro')
plot(goal(1),goal(2),'mo')

% Show start and goal headings.
r = 0.5;
plot([start(1),start(1) + r*cos(start(3))],[start(2),start(2) + r*sin(start(3))],'r-')
plot([goal(1),goal(1) + r*cos(goal(3))],[goal(2),goal(2) + r*sin(goal(3))],'m-')
hold off
```



Define State Space

Specify the state space of the vehicle using a `stateSpaceDubins` object and specifying the state bounds. This object limits the sampled states to feasible Dubins curves for steering a vehicle within the state bounds. A turning radius of 0.4 meters allows for tight turns in this small environment.

```
bounds = [occGrid.XWorldLimits; occGrid.YWorldLimits; [-pi pi]];
```

```
ss = stateSpaceDubins(bounds);
ss.MinTurningRadius = 0.4;
```

Plan The Path

To plan a path, the RRT algorithm samples random states within the state space and attempts to connect a path. These states and connections need to be validated or excluded based on the map constraints. The vehicle must not collide with obstacles defined in the map.

Create a `validatorOccupancyMap` object with the specified state space. Set the `Map` property to the loaded `occupancyMap` object. Set a `ValidationDistance` of 0.05 m. This validation distance discretizes the path connections and checks obstacles in the map based on this.

```
stateValidator = validatorOccupancyMap(ss);
stateValidator.Map = occGrid;
stateValidator.ValidationDistance = 0.05;
```

Create the path planner and increase the max connection distance to connect more states. Set the maximum number of iterations for sampling states.

```

planner = plannerRRT(ss,stateValidator);
planner.MaxConnectionDistance = 2.0;
planner.MaxIterations = 30000;

```

Customize the GoalReached function. This example helper function checks if a feasible path reaches the goal within a set threshold. The function returns true when the goal has been reached, and the planner stops.

```

planner.GoalReachedFcn = @exampleHelperCheckIfGoal;

function isReached = exampleHelperCheckIfGoal(planner, goalState, newState)
    isReached = false;
    threshold = 0.1;
    if planner.StateSpace.distance(newState, goalState) < threshold
        isReached = true;
    end
end

```

Plan the path between the start and goal. Reset the random number generator for reproducible results.

```

rng default

[pthObj, solnInfo] = plan(planner,start,goal);

```

Plot the Path

Show the occupancy map. Plot the search tree from the solnInfo. Interpolate and overlay the final path.

```

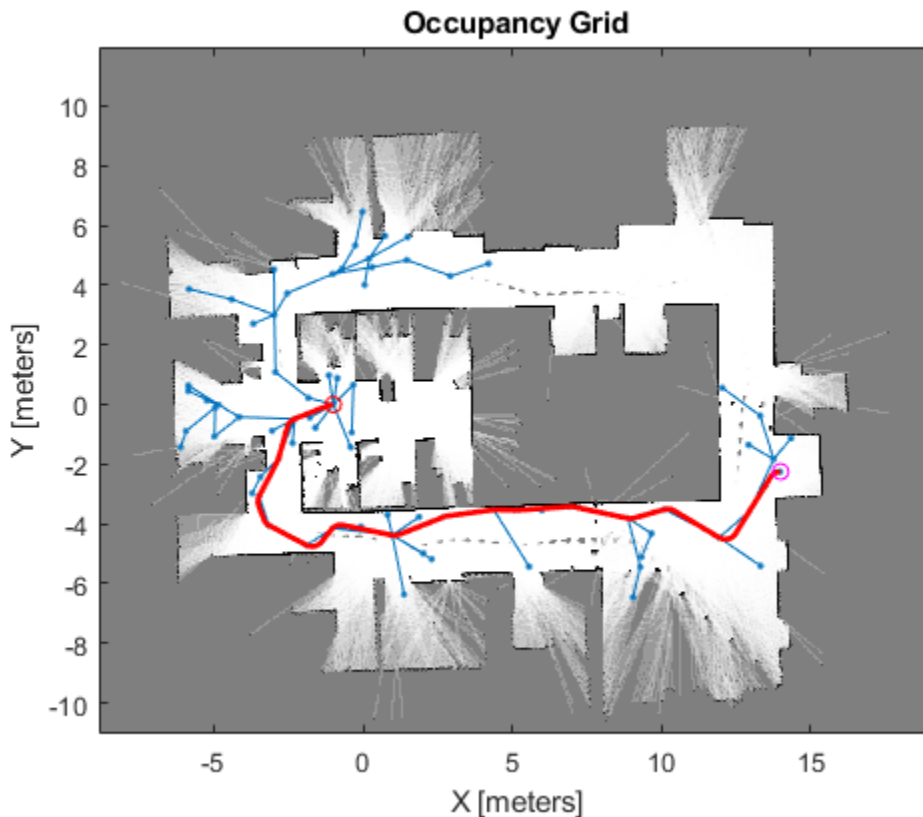
show(occGrid)
hold on

% Plot entire search tree.
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-');

% Interpolate and plot path.
interpolate(pthObj,300)
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2)

% Show start and goal in grid map.
plot(start(1),start(2),'ro')
plot(goal(1),goal(2),'mo')
hold off

```



Customize Dubins Vehicle Constraints

To specify custom vehicle constraints, customize the state space object. This example uses `ExampleHelperStateSpaceOneSidedDubins`, which is based on the `stateSpaceDubins` class. This helper class limits the turning direction to either right or left based on a Boolean property, `GoLeft`. This property essentially disables path types of the `dubinsConnection` object.

Create the state space object using the example helper. Specify the same state bounds and give the new Boolean parameter as `true` (left turns only).

```
% Only making left turns
goLeft = true;

% Create the state space
ssCustom = ExampleHelperStateSpaceOneSidedDubins(bounds,goLeft);
ssCustom.MinTurningRadius = 0.4;
```

Plan Path

Create a new planner object with the custom Dubins constraints and a validator based on those constraints. Specify the same `GoalReached` function.

```
stateValidator2 = validatorOccupancyMap(ssCustom);
stateValidator2.Map = occGrid;
stateValidator2.ValidationDistance = 0.05;

planner = plannerRRT(ssCustom,stateValidator2);
```



```
planner.MaxConnectionDistance = 2.0;
planner.MaxIterations = 30000;
planner.GoalReachedFcn = @exampleHelperCheckIfGoal;
```

Plan the path between the start and goal. Reset the random number generator again.

```
rng default
[pthObj2,solnInfo] = plan(planner,start,goal);
```

Plot Path

Draw the new path on the map. The path should only execute left turns to reach the goal.

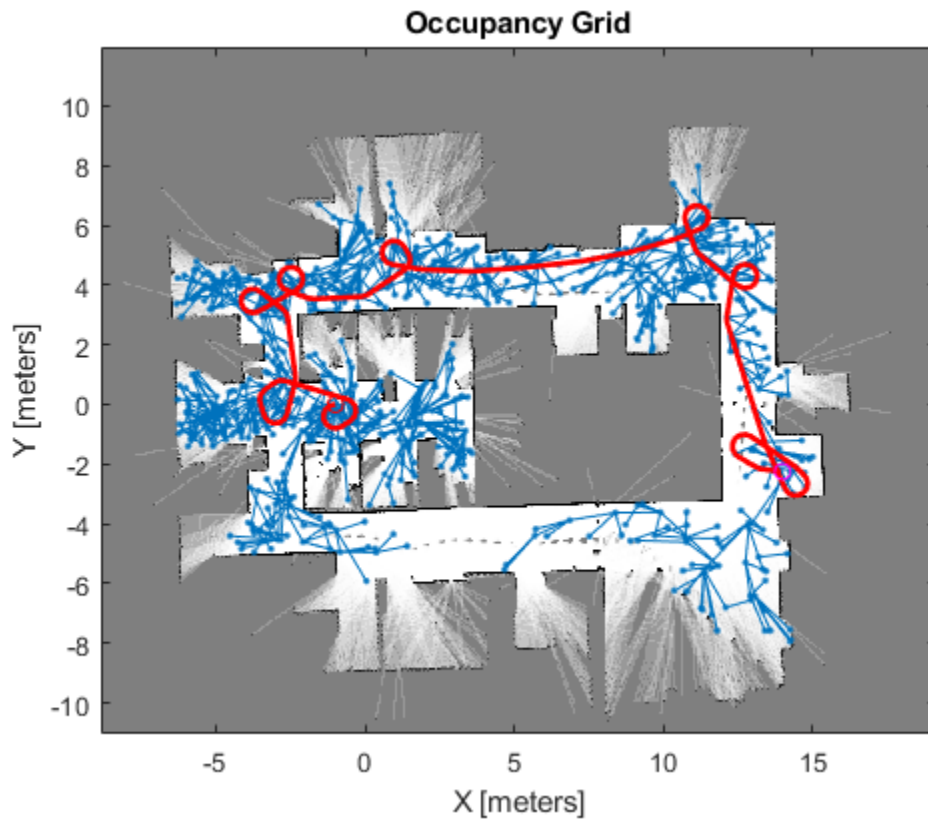
```
figure
show(occGrid)

hold on

% Show the search tree.
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-');

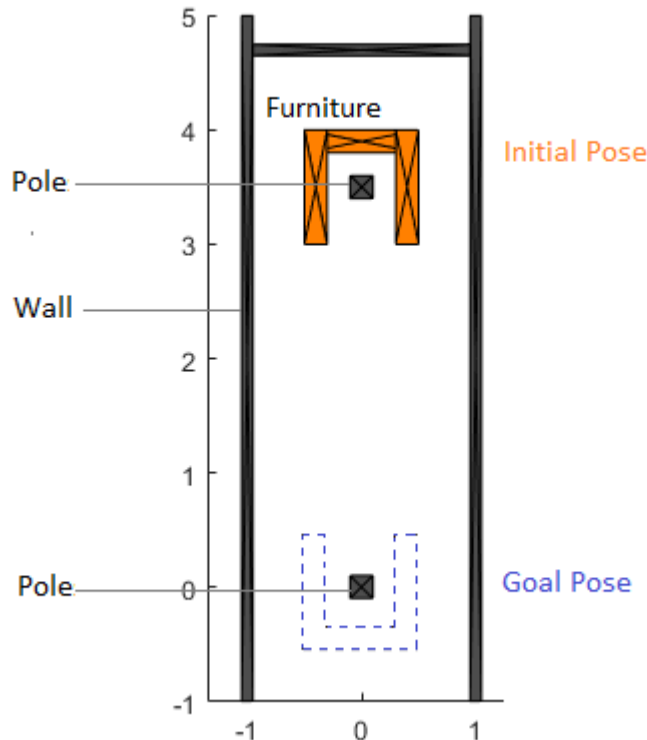
% Interpolate and plot path.
pthObj2.interpolate(300)
plot(pthObj2.States(:,1), pthObj2.States(:,2), 'r-', 'LineWidth', 2)

% Show start and goal in grid map.
plot(start(1), start(2), 'ro')
plot(goal(1), goal(2), 'mo')
hold off
```



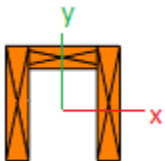
Moving Furniture in a Cluttered Room with RRT

This example shows how to plan a path to move bulky furniture in a tight space avoiding poles. This example shows a workflow of the "Piano Mover's Problem", which is used for testing path planning algorithms with constrained state spaces. This example uses the `plannerRRTStar` object to implement a custom optimized rapidly-exploring tree (RRT*) algorithm. Provided example helpers illustrate how to define custom state spaces and state validation for any motion planning application.



Model the Scene

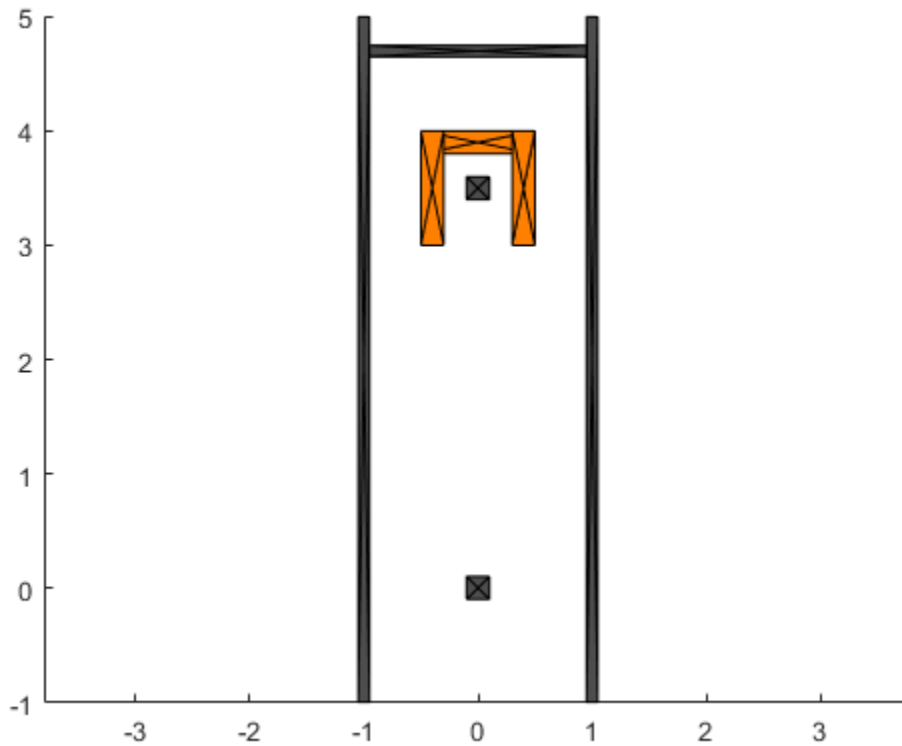
To help visualize and solve this path planning problem, two helper classes are provided, `ExampleHelperFurniture` and `ExampleHelperRoom`. The `ExampleHelperFurniture` class assembles the furniture by putting three rectangle collision boxes together. The furniture is set up as below:



The `ExampleHelperRoom` defines the dimension of the room and provides functions to insert furniture into the room and to check whether the furniture is in collision with the walls or poles. These two classes are used for the state validator of the planner.

Show the room with a given length and width. Add a piece of furniture with a given pose.

```
len = 6;  
wid = 2;  
room = ExampleHelperRoom(len, wid);  
chair = ExampleHelperFurniture;  
addFurniture(room, chair, trvec2tform([0 3.5 0]));  
show(room, gca)  
axis equal
```



Configure State Space

Create a `stateSpaceSE2` object for the furniture. Set the bounds based on the room dimensions. The state space samples random states in the state space. In this example, the furniture state is a 3-element vector, $[x \ y \ \theta]$, for the xy -coordinates and angle of rotation in radians.

```
bounds = [-0.8 0.8; [-1 5]; [-pi pi]];
```

```
ss = stateSpaceSE2(bounds);  
ss.WeightTheta = 2;
```

Create a Custom State Validator

The planner requires a customized state validator to enable collision checking between furniture and the fixtures in the room. The provided class, `ExampleHelperFurnitureInRoomValidator`, checks the validity of the furniture states based on the pairwise convex polygon collision checking functions. The class automatically creates a room and puts the weird-shaped furniture in it at construction.

```
% Set the initial pose of the furniture  
initPose = trvec2tform([0 3.5 0]);
```

```

% Create a customized state validator
sv = ExampleHelperFurnitureInRoomValidator(ss, initPose);

% Reduce the validation distance
% Validation distance determines the granularity of interpolation when
% checking the motion that connects two states.
sv.ValidationDistance = 0.1;

```

Configure the Path Planner

Use `plannerRRTStar` as the planner and specify the custom state space and state validator. Specify additional parameters for the planner. The `GoalReached` example helper function returns `true` when a feasible path gets close enough to the goal within a threshold. This exits the planner.

```

% Create the planner
rrt = plannerRRTStar(ss, sv);

% Set ball radius for searching near neighbors
rrt.BallRadiusConstant = 1000;

% Exit as soon as a path is found
rrt.ContinueAfterGoalReached = false;

% The motion length between two furniture poses should be less than 0.4 m
rrt.MaxConnectionDistance = 0.4;

% Increase the max iterations
rrt.MaxIterations = 20000;

% Use a customized goal function
rrt.GoalReachedFcn = @exampleHelperGoalFunc;

```

Plan the Move

Set a start and end pose for the furniture. This example moves from one pole to the other and rotates the chair π radians. Plan the path between poses. Visualize the search tree and final path.

```

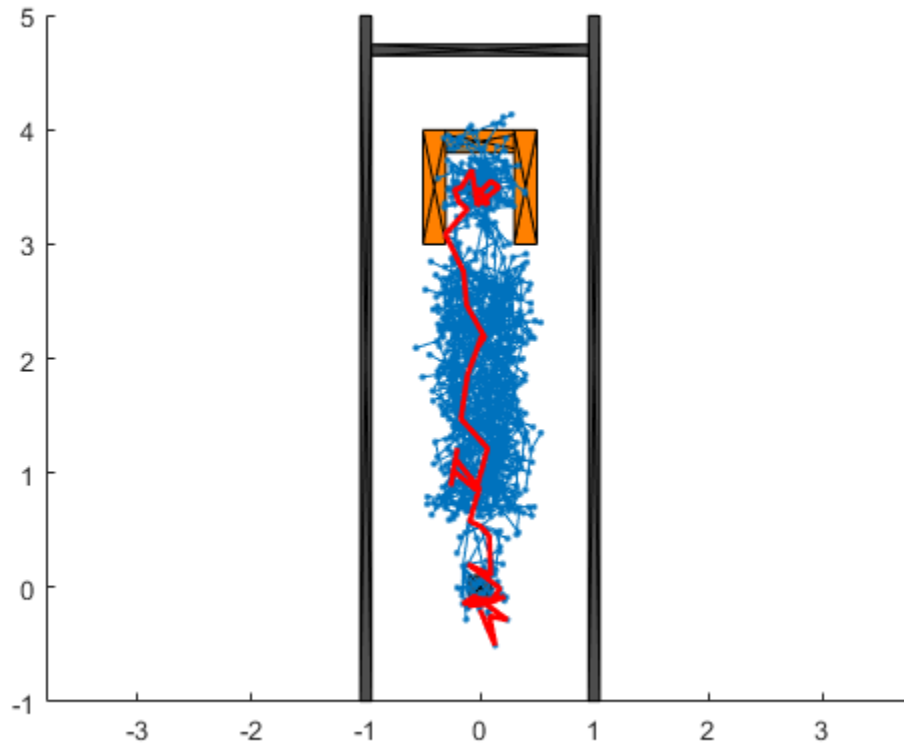
% Set the init and goal poses
start = [0 3.5 0];
goal = [0 -0.2 pi];

% Set random number seed for repeatability
rng(0, 'twister');
[path, solnInfo] = plan(rrt,start,goal);

hold on
% Search tree
plot(solnInfo.TreeData(:,1), solnInfo.TreeData(:,2), 'r-');
% Interpolate path and plot points
interpolate(path,300)
plot(path.States(:,1), path.States(:,2), 'r-', 'LineWidth', 2)

hold off

```



Visualize the Motion

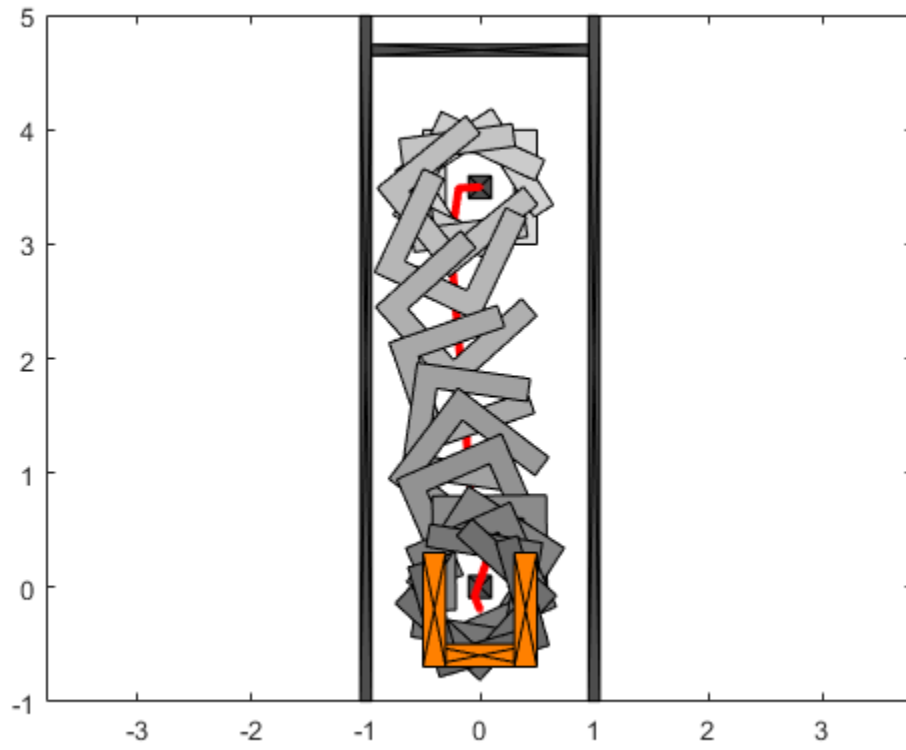
An example helper is provided for smoothing the path by cutting corners of the path where possible. Animate the motion of the furniture from start to goal pose. The animation plot shows intermediate states as the furniture navigates to the goal position.

```
f = figure;

% Smooth the path, cut the corners wherever possible
pathSm = exampleHelperSmoothPath(path, sv);

interpolate(pathSm, 100);
animateFurnitureMotion(sv.Room, 1, pathSm.States, axes(f))

% show the trace of furniture
skip = 6;
states = pathSm.States([1:skip:end, pathSm.NumStates], :);
exampleHelperShowFurnitureTrace(sv.Room.FurnituresInRoom{1}, states);
```



Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map

This example shows you how to perform dynamic replanning in an urban driving scene using a Frenet reference path. In this example, you use a dynamic occupancy grid map estimate of the local environment to find optimal local trajectories.

Introduction

Dynamic replanning for autonomous vehicles is typically done with a local motion planner. The local motion planner is responsible for generating an optimal trajectory based on the global plan and information about the surrounding environment. Information about the surrounding environment can be described mainly in two ways:

- 1 Discrete set of objects in the surrounding environment with defined geometries.
- 2 Discretized grid with estimate about free and occupied regions in the surrounding environment.

In the presence of dynamic obstacles in the environment, a local motion planner requires short-term predictions of the information about the surroundings to assess the validity of the planned trajectories. The choice of environment representation is typically governed by the upstream perception algorithm. For planning algorithms, the object-based representation offers a memory-efficient description of the environment. It also allows for an easier way to define inter-object relations for behavior prediction. On the other hand, a grid-based approach allows for an object-model-free representation, which assists in efficient collision-checking in complex scenarios with large number of objects. The grid-based representation is also less sensitive to imperfections of object extraction such as false and missed targets. A hybrid of these two approaches is also possible by extracting object hypothesis from the grid-based representation.

In this example, you represent the surrounding environment as a dynamic occupancy grid map. For an example using the discrete set of objects, refer to the “Highway Trajectory Planning Using Frenet Reference Path” on page 1-397 example. A dynamic occupancy grid map is a grid-based estimate of the local environment around the ego vehicle. In addition to estimating the probability of occupancy, the dynamic occupancy grid also estimates the kinematic attributes of each cell, such as velocity, turn-rate, and acceleration. Further, the estimates from the dynamic grid can be predicted for a short-time in the future to assess the occupancy of the local environment in the near future. In this example, you obtain the grid-based estimate of the environment by fusing point clouds from six lidars mounted on the ego vehicle.

Set Up Scenario and Grid-Based Tracker

The scenario used in this example represents an urban intersection scene and contains a variety of objects, including pedestrians, bicyclists, cars, and trucks. The ego vehicle is equipped with six homogenous lidar sensors, each with a field of view of 90 degrees, providing 360-degree coverage around the ego vehicle. For more details on the scenario and sensor models, refer to the “Grid-Based Tracking in Urban Environments Using Multiple Lidars” (Sensor Fusion and Tracking Toolbox) example. The definition of scenario and sensors is wrapped in the helper function `helperGridBasedPlanningScenario`.

```
% For reproducible results  
rng(2020);
```

```
% Create scenario, ego vehicle and simulated lidar sensors  
[scenario, egoVehicle, lidars] = helperGridBasedPlanningScenario;
```


Now, define a grid-based tracker using the `trackerGridRFS` (Sensor Fusion and Tracking Toolbox) System object™. The tracker outputs both object-level and grid-level estimate of the environment. The grid-level estimate describes the occupancy and state of the local environment and can be obtained as the fourth output from the tracker. For more details on how to set up a grid-based tracker, refer to the “Grid-Based Tracking in Urban Environments Using Multiple Lidars” (Sensor Fusion and Tracking Toolbox) example.

```
% Set up sensor configurations for each lidar
sensorConfigs = cell(numel(lidars),1);

% Fill in sensor configurations
for i = 1:numel(sensorConfigs)
    sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
end

% Set up tracker
tracker = trackerGridRFS('SensorConfigurations',sensorConfigs,...
    'HasSensorConfigurationsInput',true,...
    'GridLength',120,...
    'GridWidth',120,...
    'GridResolution',2,...
    'GridOriginInLocal',[-60 -60],...
    'NumParticles',1e5,...
    'NumBirthParticles',2e4,...
    'VelocityLimits',[-15 15;-15 15],...
    'BirthProbability',0.025,...
    'ProcessNoise',5*eye(2),...
    'DeathRate',1e-3,...
    'FreeSpaceDiscountFactor',1e-2,...
    'AssignmentThreshold',8,...
    'MinNumCellsPerCluster',4,...
    'ClusteringThreshold',4,...
    'ConfirmationThreshold',[3 4],...
    'DeletionThreshold',[4 4]);
```

Set Up Motion Planner

Set up a local motion planning algorithm to plan optimal trajectories in Frenet coordinates along a global reference path.

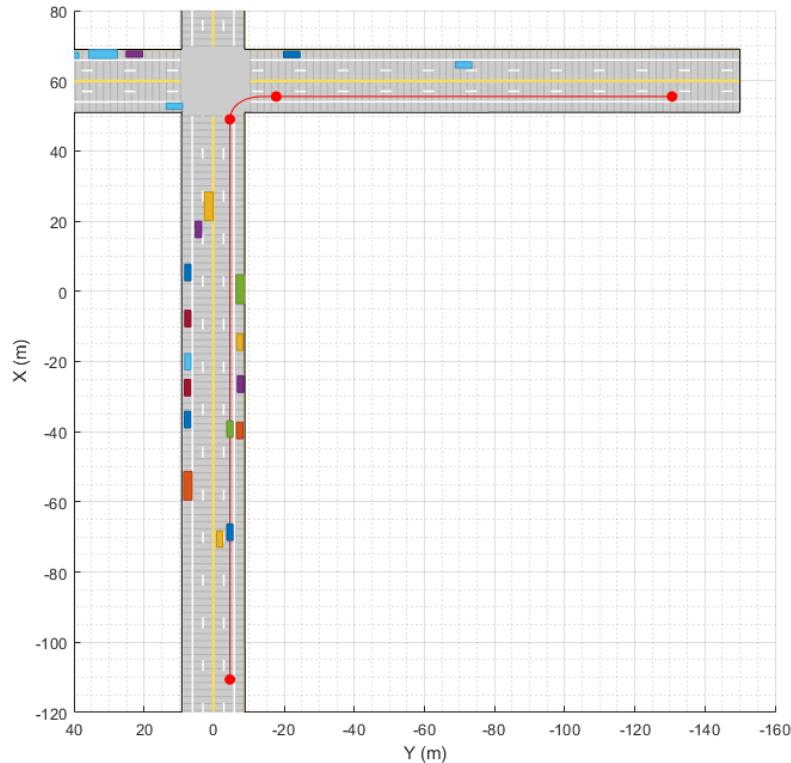
Define the global reference path using the `referencePathFrenet` object by providing the waypoints in the Cartesian coordinate frame of the driving scenario. The reference path used in this example defines a path that turns right at the intersection.

```
waypoints = [-110.6 -4.5 0;
            49 -4.5 0;
            55.5 -17.7 -pi/2;
            55.5 -130.6 -pi/2]; % [x y theta]

% Create a reference path using waypoints
refPath = referencePathFrenet(waypoints);

% Visualize the reference path
fig = figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
ax = axes(fig);
hold(ax,'on');
plot(scenario,'Parent',ax);
```

```
show(refPath, 'Parent', ax);
xlim(ax, [-120 80]);
ylim(ax, [-160 40]);
```



```
snapnow;
```

The local motion planning algorithm in this example consists of three main steps:

- 1 Sample local trajectories
- 2 Find feasible and collision-free trajectories
- 3 Choose optimality criterion and select optimal trajectory

The following sections discuss each step of the local planning algorithm and the helper functions used to execute each step.

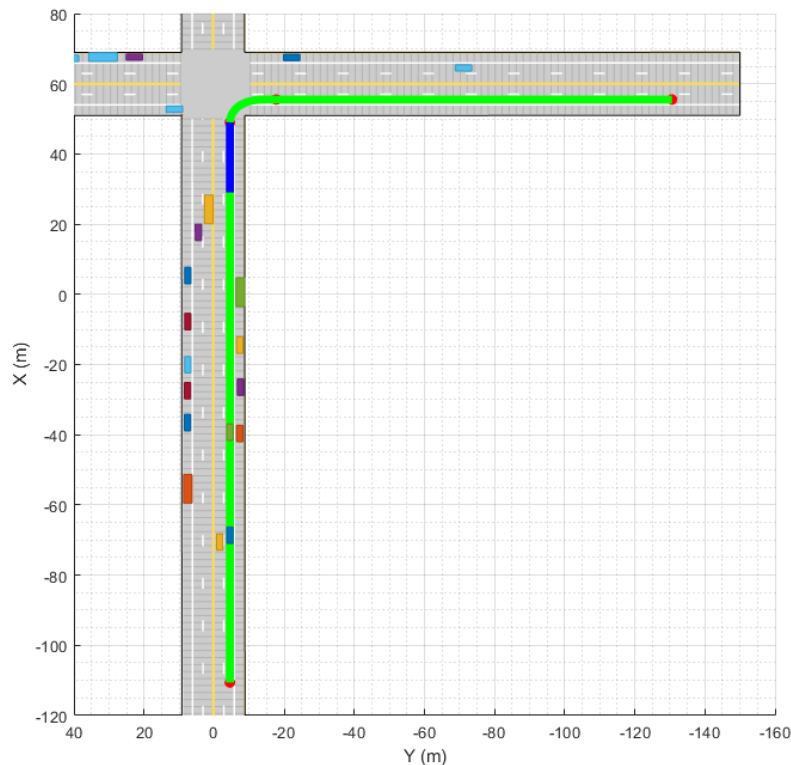
Sample Local Trajectories

At each step of the simulation, the planning algorithm generates a list of sample trajectories that the ego vehicle can choose. The local trajectories are sampled by connecting the current state of the ego vehicle to desired terminal states. Use the `trajectoryGeneratorFrenet` object to connect current and terminal states for generating local trajectories. Define the object by providing the reference path and the desired resolution in time for the trajectory. The object connects initial and final states in Frenet coordinates using fifth-order polynomials.

```
connector = trajectoryGeneratorFrenet(refPath, 'TimeResolution', 0.1);
```

The strategy for sampling terminal states in Frenet coordinates often depends on the road network and the desired behavior of the ego vehicle during different phases of the global path. For more detailed examples of using different ego behavior, such as cruise-control and car-following, refer to the "Planning Adaptive Routes Through Traffic" section of the "Highway Trajectory Planning Using Frenet Reference Path" on page 1-397 example. In this example, you sample the terminal states using two different strategies, depending on the location of vehicle on the reference path, shown as blue and green regions in the following figure.

```
% Visualize path regions for sampling strategy visualization
pathPoints = closestPoint(refPath, refPath.Waypoints(:,1:2));
roadS = pathPoints(:,end);
intersectionS = roadS(2,end);
intersectionBuffer = 20;
pathGreen = [interpolate(refPath,linspace(0,intersectionS-intersectionBuffer,20));...
            nan(1,6);...
            interpolate(refPath,linspace(intersectionS,roadS(end),100))];
pathBlue = interpolate(refPath,linspace(intersectionS-intersectionBuffer,roadS(2,end),20));
hold(ax,'on');
plot(ax,pathGreen(:,1),pathGreen(:,2),'Color',[0 1 0],'LineWidth',5);
plot(ax,pathBlue(:,1),pathBlue(:,2),'Color',[0 0 1],'LineWidth',5);
```



```
snapnow;
```

When the ego vehicle is in the green region, the following strategy is used to sample local trajectories. The terminal state of the ego vehicle after ΔT time is defined as:

$$x_{\text{Ego}}(\Delta T) = [\text{NaN } \dot{s} \ d \ 0 \ 0];$$

where discrete samples for variables are obtained using the following predefined sets:

$$\{\Delta T \in \{\text{linspace}(2, 4, 6)\}, \dot{s} \in \{\text{linspace}(0, \dot{s}_{\text{max}}, 10)\}, d \in \{0 \ w_{\text{lane}}\}\}$$

The use of NaN in the terminal state enables the `trajectoryGeneratorFrenet` object to automatically compute the longitudinal distance traveled over a minimum-jerk trajectory. This strategy produces a set of trajectories that enable the ego vehicle to accelerate up to the maximum speed limit (\dot{s}_{max}) rates or decelerate to a full stop at different rates. In addition, the sampled choices of lateral offset (d_{des}) allow the ego vehicle to change lanes during these maneuvers.

```
% Define smax and wlane
speedLimit = 15;
laneWidth = 2.975;
```

When the ego vehicle is in the blue region of the trajectory, the following strategy is used to sample local trajectories:

$$x_{\text{Ego}}(\Delta T) = [s_{\text{stop}} \ 0 \ 0 \ 0 \ 0];$$

where ΔT is chosen to minimize jerk during the trajectory. This strategy enables the vehicle to stop at the desired distance (s_{stop}) in the right lane with a minimum-jerk trajectory. The trajectory sampling algorithm is wrapped inside the helper function, `helperGenerateTrajectory`, attached with this example.

Finding Feasible and Collision-Free Trajectories

The sampling process described in the previous section can produce trajectories that are kinematically infeasible and exceed thresholds of kinematic attributes such as acceleration and curvature. Therefore, you limit the maximum acceleration and speed of the ego vehicle using the helper function `helperKinematicFeasibility` on page 1-0 , which checks the feasibility of each trajectory against these kinematic constraints.

```
% Define kinematic constraints
accMax = 15;
```

Further, you set up a collision-validator to assess if the ego vehicle can maneuver on a kinematically feasible trajectory without colliding with any other obstacles in the environment. To define the validator, use the helper class `HelperDynamicMapValidator`. This class uses the `predictMapToTime` (Sensor Fusion and Tracking Toolbox) function of the `trackerGridRFS` object to get short-term predictions of the occupancy of the surrounding environment. Since the uncertainty in the estimate increases with time, configure the validator with a maximum time horizon of 2 seconds.

The predicted occupancy of the environment is converted to an inflated costmap at each step to account for the size of the ego vehicle. The path planner uses a timestep of 0.1 seconds with a prediction time horizon of 2 seconds. To reduce computational complexity, the occupancy of the surrounding environment is assumed to be valid for 5 time steps, or 0.5 seconds. As a result, only 4 predictions are required in the 2-second planning horizon. In addition to making binary decisions about collision or no collision, the validator also provides a measure of collision probability of the ego vehicle. This probability can be incorporated into the cost function for optimality criteria to account for uncertainty in the system and to make better decisions without increasing the time horizon of the planner.

```

vehDims = vehicleDimensions(egoVehicle.Length,egoVehicle.Width);
collisionValidator = HelperDynamicMapValidator('MaxTimeHorizon',2, ... % Maximum horizon for val.
'TimeResolution',connector.TimeResolution, ... % Time steps between trajectory samples
'Tracker',tracker, ... % Provide tracker for prediction
'ValidPredictionSpan',5, ... % Prediction valid for 5 steps
'VehicleDimensions',vehDims); % Provide dimensions of ego

```

Choose Optimality Criterion

After validating the feasible trajectories against obstacles or occupied regions of the environment, choose an optimality criterion for each valid trajectory by defining a cost function for the trajectories. Different cost functions are expected to produce different behaviors from the ego vehicle. In this example, you define the cost of each trajectory as

$$C = J_s + J_d + 1000P_c + 100(\dot{s}_{(\Delta T)} - \dot{s}_{\text{Limit}})^2$$

where:

J_s is the jerk in the longitudinal direction of the reference path

J_d is the jerk in the lateral direction of the reference path

P_c is the collision probability obtained by the validator

The cost calculation for each trajectory is defined using the helper function `helperCalculateTrajectoryCosts` on page 1-0 . From the list of valid trajectories, the trajectory with the minimum cost is considered as the optimal trajectory.

Run Scenario, Estimate Dynamic Map, and Plan Local Trajectories

Run the scenario, generate point clouds from all the lidar sensors, and estimate the dynamic occupancy grid map. Use the dynamic map estimate and its predictions to plan a local trajectory for the ego vehicle.

```

% Close original figure and initialize a new display
close(fig);
display = helperGridBasedPlanningDisplay;

% Initial ego state
currentEgoState = [-110.6 -1.5 0 0 15 0];
helperMoveEgoVehicleToState(egoVehicle, currentEgoState);

% Initialize pointCloud outputs from each sensor
ptClouds = cell(numel(lidars),1);
sensorConfigs = cell(numel(lidars),1);

% Simulation Loop
while advance(scenario)
    % Current simulation time
    time = scenario.SimulationTime;

    % Poses of objects with respect to ego vehicle
    tgtPoses = targetPoses(egoVehicle);

    % Simulate point cloud from each sensor
    for i = 1:numel(lidars)

```

```

        [ptClouds{i}, isValidTime] = step(lidars{i},tgtPoses,time);
        sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
    end

    % Pack point clouds as sensor data format required by the tracker
    sensorData = packAsSensorData(ptClouds,sensorConfigs,time);

    % Call the tracker
    [tracks, ~, ~, map] = tracker(sensorData,sensorConfigs,time);

    % Update validator's future predictions using current estimate
    step(collisionValidator, currentEgoState, map, time);

    % Sample trajectories using current ego state and some kinematic
    % parameters
    [frenetTrajectories, globalTrajectories] = helperGenerateTrajectory(connector, refPath, currentEgoState, map, sensorData, sensorConfigs, time);

    % Calculate kinematic feasibility of generated trajectories
    isKinematicsFeasible = helperKinematicFeasibility(frenetTrajectories,speedLimit,accMax);

    % Calculate collision validity of feasible trajectories
    feasibleGlobalTrajectories = globalTrajectories(isKinematicsFeasible);
    feasibleFrenetTrajectories = frenetTrajectories(isKinematicsFeasible);
    [isCollisionFree, collisionProb] = isTrajectoryValid(collisionValidator, feasibleGlobalTrajectories, feasibleFrenetTrajectories, currentEgoState, map, time);

    % Calculate costs and final optimal trajectory
    nonCollidingGlobalTrajectories = feasibleGlobalTrajectories(isCollisionFree);
    nonCollidingFrenetTrajectories = feasibleFrenetTrajectories(isCollisionFree);
    nonCollidingCollisionProb = collisionProb(isCollisionFree);
    costs = helperCalculateTrajectoryCosts(nonCollidingFrenetTrajectories, nonCollidingCollisionProb, currentEgoState, map, time);

    % Find optimal trajectory
    [~,idx] = min(costs);
    optimalTrajectory = nonCollidingGlobalTrajectories(idx);

    % Assemble for plotting
    trajectories = helperAssembleTrajectoryForPlotting(globalTrajectories, ...
        isKinematicsFeasible, isCollisionFree, idx);

    % Update display
    display(scenario, egoVehicle, lidars, ptClouds, tracker, tracks, trajectories, collisionValidator);

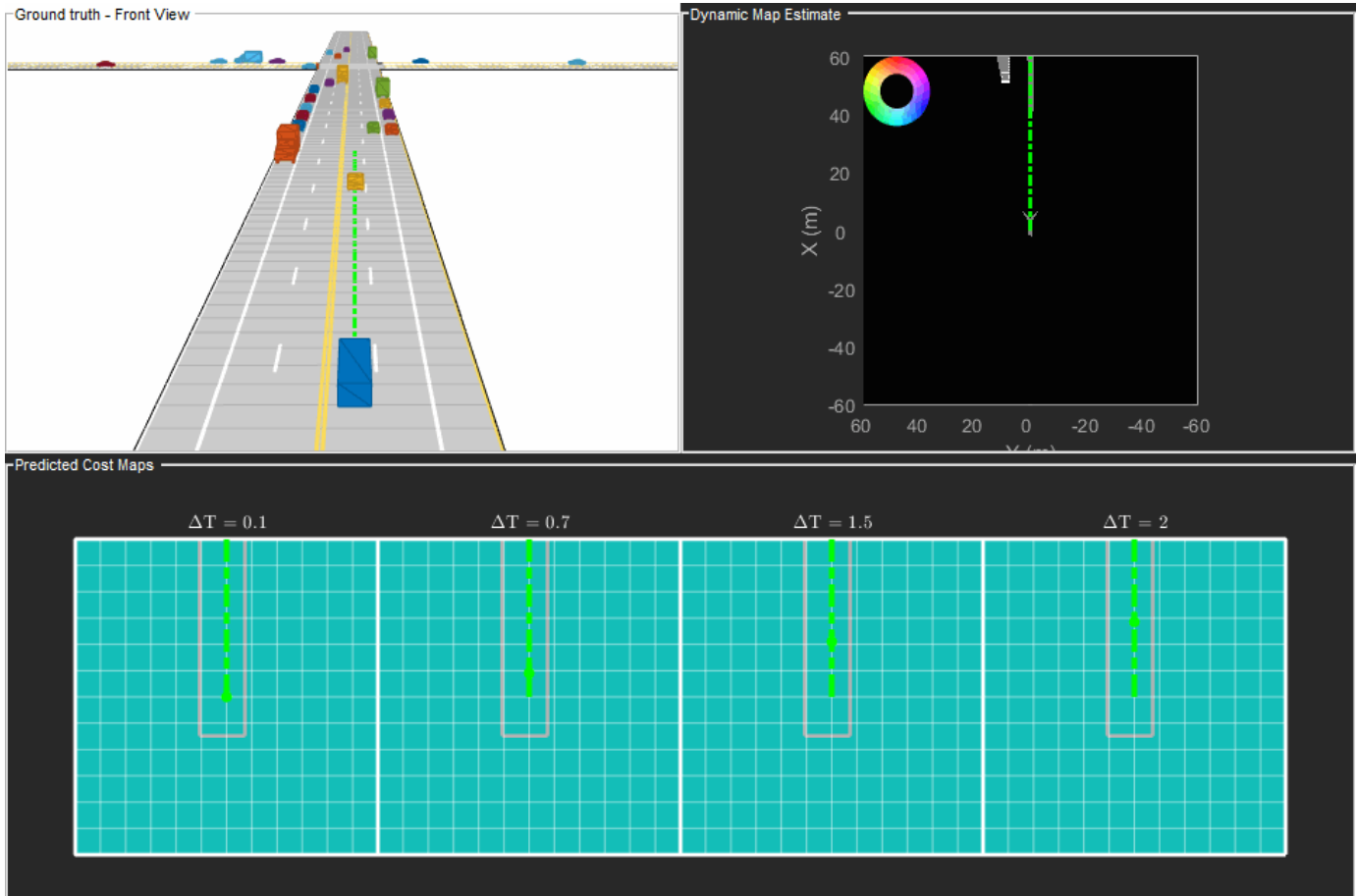
    % Move ego with optimal trajectory
    if ~isempty(optimalTrajectory)
        currentEgoState = optimalTrajectory.Trajectory(2,:);
        helperMoveEgoVehicleToState(egoVehicle, currentEgoState);
    else
        % All trajectories either violated kinematic feasibility
        % constraints or resulted in a collision. More behaviors on
        % trajectory sampling may be needed.
        error('Unable to compute optimal trajectory');
    end
end

```

Results

Analyze the results from the local path planning algorithm and how the predictions from the map assisted the planner. This animation shows the result of the planning algorithm during the entire

scenario. Notice that the ego vehicle successfully reached its desired destination and maneuvered around different dynamic objects, whenever necessary. The ego vehicle also came to a stop at the intersection due to the regional changes added to the sampling policy.

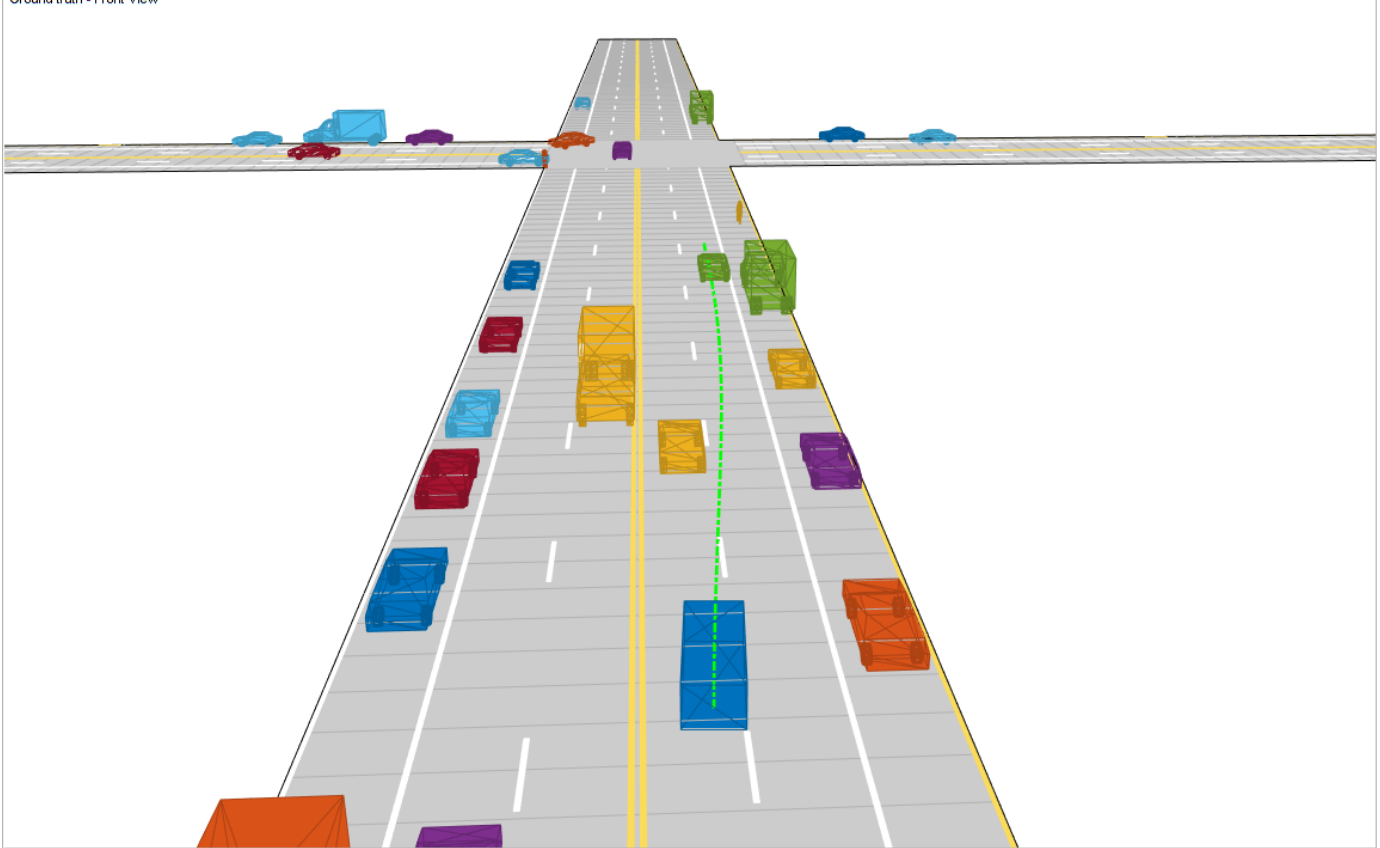


Next, analyze the local planning algorithm during the first lane change. The snapshots in this section are captured at time = 4.3 seconds during the simulation.

In this snapshot, the ego vehicle has just started to perform a lane change maneuver into the right lane.

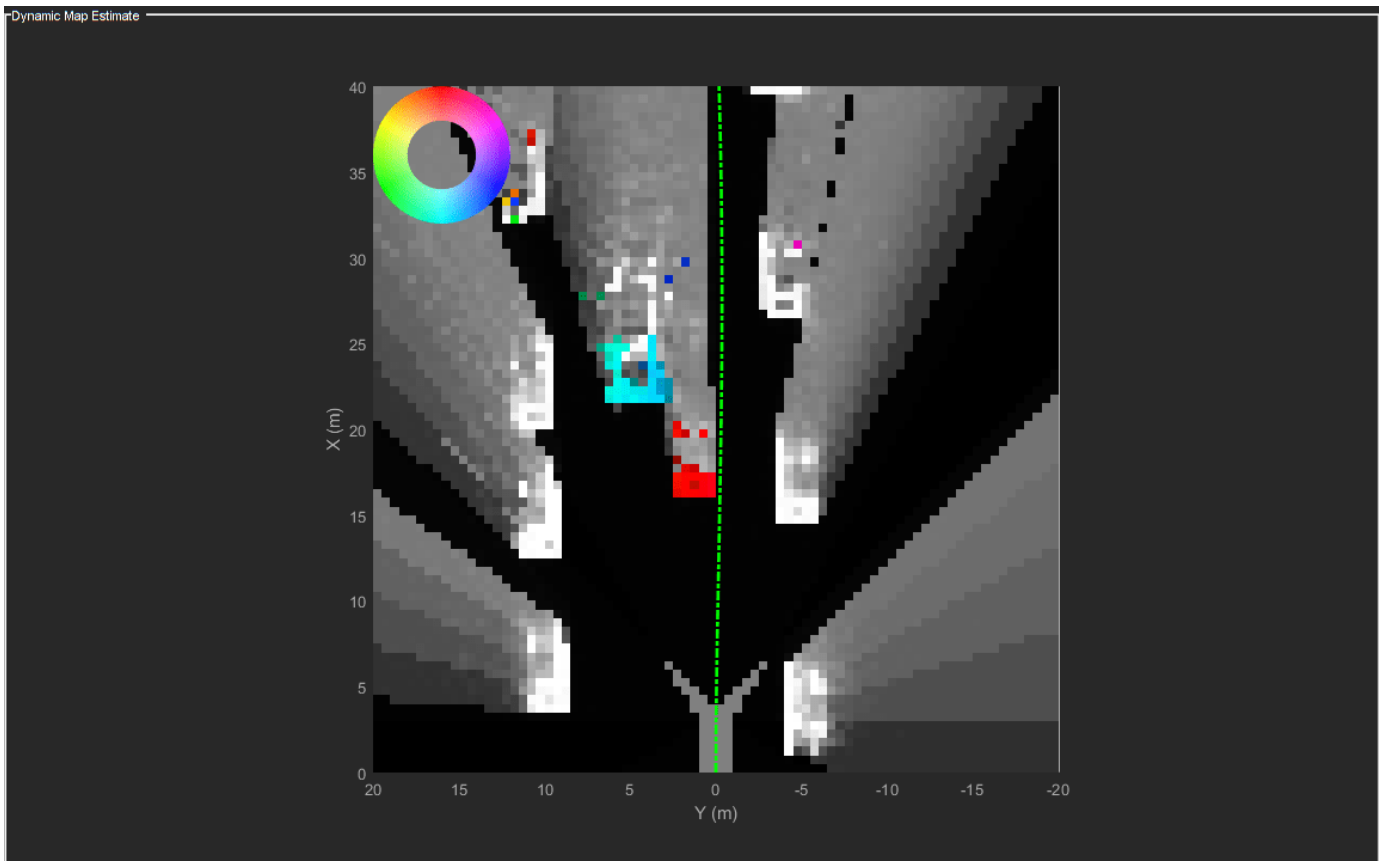
```
showSnaps(display, 3, 1);
```

Ground truth - Front View



The snapshot that follows shows the estimate of the dynamic grid at the same time step. The color of the grid cell denotes the direction of motion of the object occupying that grid cell. Notice that the cells representing the car in front of the ego vehicle are colored red, denoting that the cells are occupied with a dynamic object. Also, the car is moving in the positive X direction of the scenario, so based on the color wheel, the color of the corresponding grid cells is red.

```
f = showSnaps(display, 2, 1);
if ~isempty(f)
    ax = findall(f, 'Type', 'Axes');
    ax.XLim = [0 40];
    ax.YLim = [-20 20];
    s = findall(ax, 'Type', 'Surf');
    s.XData = 36 + 1/3*(s.XData - mean(s.XData(:)));
    s.YData = 16 + 1/3*(s.YData - mean(s.YData(:)));
end
```

Based on the previous image, the planned trajectory of the ego vehicle passes through the occupied regions of space, representing a collision if you performed a traditional static occupancy validation. The dynamic occupancy map and the validator, however, account for the dynamic nature of the grid by validating the state of the trajectory against the predicted occupancy at each time step. The next snapshot shows the predicted costmap at different prediction steps (ΔT), along with the planned position of the ego vehicle on the trajectory. The predicted costmap is inflated to account for size of the ego vehicle. Therefore, if a point object representing the origin of the ego vehicle can be placed on the occupancy map without any collision, it can be interpreted that the ego vehicle does not collide with any obstacle. The yellow regions on the costmap denote areas with guaranteed collisions with an obstacle. The collision probability decays outside the yellow regions exponentially until the end of inflation region. The blue regions indicate areas with zero probability of collision according to the current prediction.

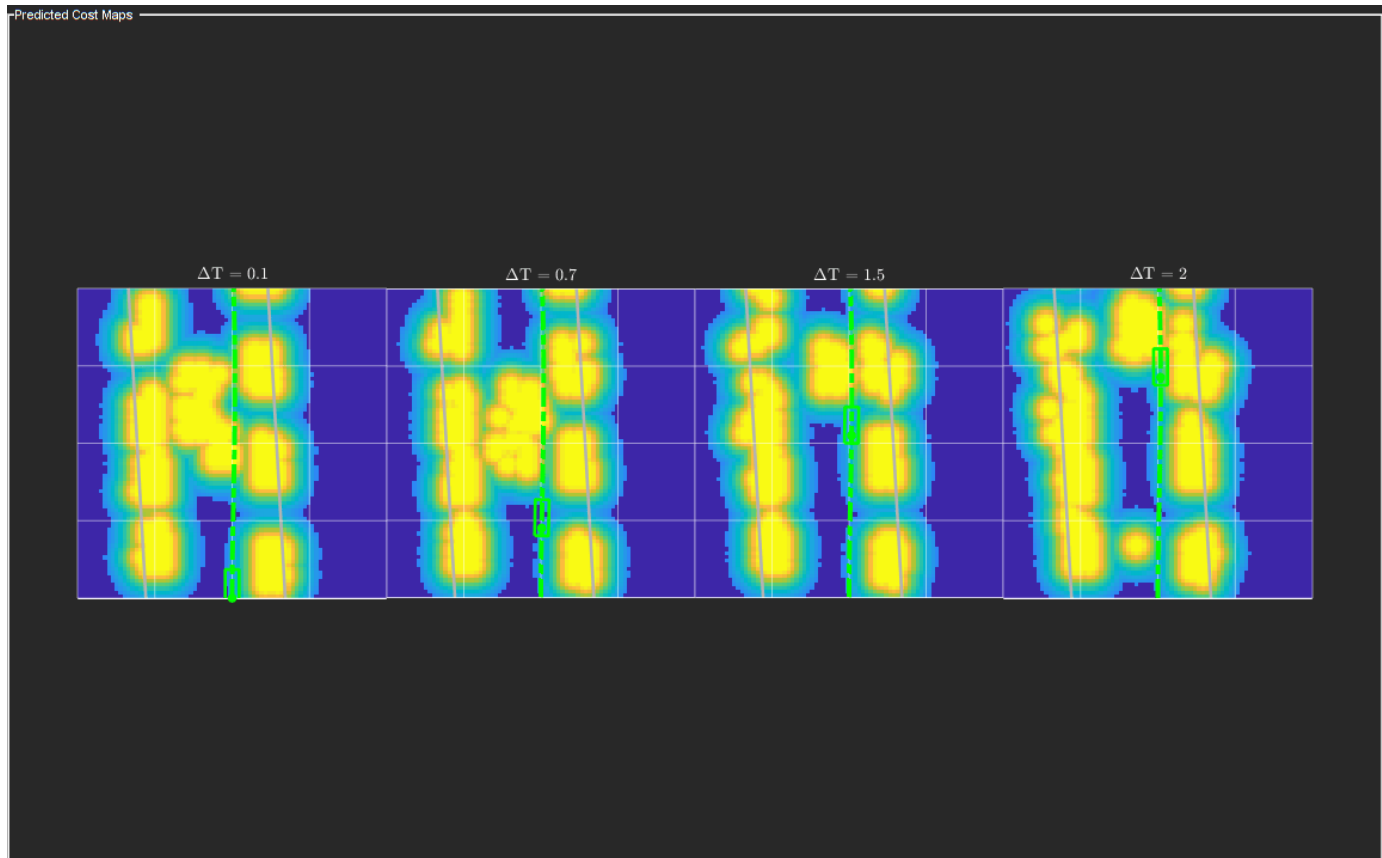
Notice that the yellow region representing the car in front of the ego vehicle moves forward on the costmap as the map is predicted in the future. This reflects that the prediction of occupancy considers the velocity of objects in the surrounding environment. Also, notice that the cells classified as static objects remained relatively static on the grid during the prediction. Lastly, notice that the planned position of the ego vehicle origin does not collide with any occupied regions in the cost map. This shows that the ego vehicle can successfully maneuver on this trajectory.

```
f = showSnaps(display, 1, 1);
if ~isempty(f)
ax = findall(f, 'Type', 'Axes');
for i = 1:numel(ax)
    ax(i).XLim = [0 40];
end
```

```

ax(i).YLim = [-20 20];
end
end

```



Summary

In this example, you learned how to use the dynamic map predictions from the grid-based tracker, `trackerGridRFS`, and how to integrate the dynamic map with a local path planning algorithm to generate trajectories for the ego vehicle in dynamic complex environments. You also learned how the dynamic nature of the occupancy can be used to plan trajectories more efficiently in the environment.

Supporting Functions

```

function sensorData = packAsSensorData(ptCloud, configs, time)
% Pack the sensor data as format required by the tracker
%
% ptCloud - cell array of pointCloud object
% configs - cell array of sensor configurations
% time    - Current simulation time

%The lidar simulation returns outputs as pointCloud objects. The Location
%property of the point cloud is used to extract x,y, and z locations of
%returns and pack them as structures with information required by a tracker.
sensorData = struct('SensorIndex', {}, ...
    'Time', {}, ...
    'Measurement', {}, ...
    'MeasurementParameters', {});

```

```

for i = 1:numel(ptCloud)
    % This sensor's point cloud
    thisPtCloud = ptCloud{i};

    % Allows mapping between data and configurations without forcing an
    % ordered input and requiring configuration input for static sensors.
    sensorData(i).SensorIndex = configs{i}.SensorIndex;

    % Current time
    sensorData(i).Time = time;

    % Extract Measurement as a 3-by-N defining locations of points
    sensorData(i).Measurement = reshape(thisPtCloud.Location,[],3)';

    % Data is reported in the sensor coordinate frame and hence measurement
    % parameters are same as sensor transform parameters.
    sensorData(i).MeasurementParameters = configs{i}.SensorTransformParameters;
end

end

function config = helperGetLidarConfig(lidar, ego)
% Get configuration of the lidar sensor for tracker
%
% config - Configuration of the lidar sensor in the world frame
% lidar - lidarPointCloudGeneration object
% ego    - driving.scenario.Actor in the scenario

% Define transformation from sensor to ego
senToEgo = struct('Frame',fusionCoordinateFrameType(1),...
    'OriginPosition',[lidar.SensorLocation(:);lidar.Height],...
    'Orientation',rotmat(Quaternion([lidar.Yaw lidar.Pitch lidar.Roll],'eulerd','ZYX','frame'),'frame'),...
    'IsParentToChild',true);

% Define transformation from ego to tracking coordinates
egoToScenario = struct('Frame',fusionCoordinateFrameType(1),...
    'OriginPosition',ego.Position(:),...
    'Orientation',rotmat(Quaternion([ego.Yaw ego.Pitch ego.Roll],'eulerd','ZYX','frame'),'frame'),...
    'IsParentToChild',true);

% Assemble using trackingSensorConfiguration.
config = trackingSensorConfiguration(...
    'SensorIndex',lidar.SensorIndex,...
    'IsValidTime', true,...
    'SensorLimits',[lidar.AzimuthLimits;0 lidar.MaxRange],...
    'SensorTransformParameters',[senToEgo;egoToScenario],...
    'DetectionProbability',0.95);

end

function helperMoveEgoVehicleToState(egoVehicle, currentEgoState)
% Move ego vehicle in scenario to a state calculated by the planner
%
% egoVehicle - driving.scenario.Actor in the scenario
% currentEgoState - [x y theta kappa speed acc]

% Set 2-D Position

```

```

egoVehicle.Position(1:2) = currentEgoState(1:2);

% Set 2-D Velocity (s*cos(yaw) s*sin(yaw))
egoVehicle.Velocity(1:2) = [cos(currentEgoState(3)) sin(currentEgoState(3))]*currentEgoState(5);

% Set Yaw in degrees
egoVehicle.Yaw = currentEgoState(3)*180/pi;

% Set angular velocity in Z (yaw rate) as v/r
egoVehicle.AngularVelocity(3) = currentEgoState(4)*currentEgoState(5);

end

function isFeasible = helperKinematicFeasibility(frenetTrajectories, speedLimit, aMax)
% Check kinematic feasibility of trajectories
%
% frenetTrajectories - Array of trajectories in Frenet coordinates
% speedLimit - Speed limit (m/s)
% aMax - Maximum acceleration (m/s^2)

isFeasible = false(numel(frenetTrajectories),1);
for i = 1:numel(frenetTrajectories)
    % Speed of the trajectory
    speed = frenetTrajectories(i).Trajectory(:,2);

    % Acceleration of the trajectory
    acc = frenetTrajectories(i).Trajectory(:,3);

    % Is speed valid?
    isSpeedValid = ~any(speed < -0.1 | speed > speedLimit + 1);

    % Is acceleration valid?
    isAccelerationValid = ~any(abs(acc) > aMax);

    % Trajectory feasible if both speed and acc valid
    isFeasible(i) = isSpeedValid & isAccelerationValid;
end

end

function cost = helperCalculateTrajectoryCosts(frenetTrajectories, Pc, smax)
% Calculate cost for each trajectory.
%
% frenetTrajectories - Array of trajectories in Frenet coordinates
% Pc - Probability of collision for each trajectory calculated by validator

n = numel(frenetTrajectories);
Jd = zeros(n,1);
Js = zeros(n,1);
s = zeros(n,1);

for i = 1:n
    % Time
    time = frenetTrajectories(i).Times;

    % resolution
    dT = time(2) - time(1);

```

```
% Jerk along the path
dds = frenetTrajectories(i).Trajectory(:,3);
Js(i) = sum(gradient(dds,time).^2)*dT;

% Jerk perpendicular to path
% d2L/dt2 = d/dt(dL/ds*ds/dt)
ds = frenetTrajectories(i).Trajectory(:,2);
ddL = frenetTrajectories(i).Trajectory(:,6).*(ds.^2) + frenetTrajectories(i).Trajectory(:,5)
Jd(i) = sum(gradient(ddL,time).^2)*dT;

s(i) = frenetTrajectories(i).Trajectory(end,2);
end

cost = Js + Jd + 1000*Pc(:) + 100*(s - smax).^2;
end
```

Motion Planning with RRT for a Robot Manipulator

Plan a grasping motion for a **Kinova Jaco Assistive Robotics Arm** using the rapidly-exploring random tree (RRT) algorithm. This example uses a `plannerRRTStar` object to sample states and plan the robot motion. Provided example helpers illustrate how to define custom state spaces and state validation for motion planning applications.

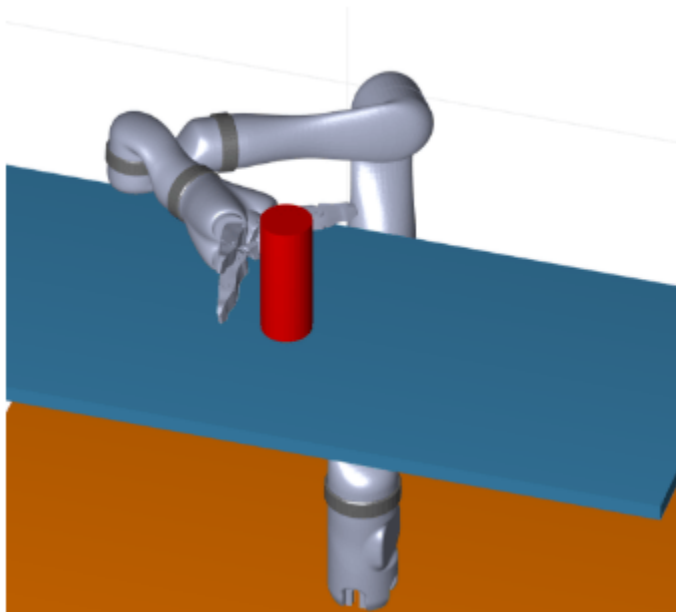
Load a Kinova Jaco model from the robot library. This particular model includes the three-finger gripper.

```
kin = loadrobot('kinovaJacoJ2S7S300');
```

Create The Environment

Using collision object primitives, add a floor, two table tops, and a cylinder can. Specify the size and pose of these objects. The provided image shows the environment created.

```
floor = collisionBox(1, 1, 0.01);
tabletop1 = collisionBox(0.4,1,0.02);
tabletop1.Pose = trvec2tform([0.3,0,0.6]);
tabletop2 = collisionBox(0.6,0.2,0.02);
tabletop2.Pose = trvec2tform([-0.2,0.4,0.5]);
can = collisionCylinder(0.03,0.16);
can.Pose = trvec2tform([0.3,0.0,0.7]);
```



Customize The State Space For Manipulator

The Kinova arm has ten degrees of freedom (DoFs), with the last three DoFs corresponding to the fingers. Only use the first seven DoFs for the planning and keep the fingers at zero configuration (open wide). An `ExampleHelperRigidBodyTreeStateSpace` state space is created to represent the configuration space (joint space). `ExampleHelperRigidBodyTreeStateSpace` samples feasible states for the robot arm. The `sampleUniform` function of the state space alternates between the following two sampling strategies with equal probability:

- Uniformly random sample the end effector pose in the **Workspace Goal Region** around the reference goal pose, then map it to the joint space through inverse kinematics. Joint limits are respected.
- Uniformly random sample in the joint space. Joint limits are respected.

The first sampling strategy helps guide the RRT planner towards the goal region in the task space so that RRT can converge to a solution faster instead of getting lost in the seven DoF joint space.

Using **Workspace Goal Region** (WGR) instead of single goal pose increases the chance of finding a solution by biasing samples to the goal region. WGR defines a continuum of acceptable end-effector poses for certain tasks. For example, the robot can approach from multiple directions to grasp a cup of water from the side, as long as it doesn't collide with the environment. The concept of WGR is first proposed by Dmitry Berenson et al [1] in 2009. This algorithm later evolved into **Task Space Regions** [2]. A WGR consists of three parts:

- `Twgr_0` - The reference transform of a WGR in world ($\{0\}$) coordinates
- `Te_w` - The end-effector offset transform in the $\{w\}$ coordinates, $\{w\}$ is sampled from WGR
- `Bounds` - A 6-by-2 matrix of bounds in the WGR reference coordinates. The first three rows of `Bounds` set the allowable translation along the x, y, and z axes (in meters) respectively and the last three set the allowable rotations about the allowable rotations about the x, y, and z axes (in radians). Note that the Roll-Pitch-Yaw (RPY) Euler angles are used as they can be intuitively specified.

You can define and concatenate multiple WGRs in one planning problem. In this example, only one WGR is allowed.

```
% Create state space and set workspace goal regions (WGRs)
ss = ExampleHelperRigidBodyTreeStateSpace(kin);
ss.EndEffector = 'j2s7s300_end_effector';

% Define the workspace goal region (WGR)
% This WGR tells the planner that the can shall be grasped from
% the side and the actual grasp height may wiggle at most 1 cm.

% This is the orientation offset between the end-effector in grasping pose and the can frame
R = [0 0 1; 1 0 0; 0 1 0];

Tw_0 = can.Pose;
Te_w = rotm2tform(R);
bounds = [0 0;           % x
          0 0;           % y
          0 0.01;       % z
          0 0;           % R
          0 0;           % P
          -pi pi];      % Y
setWorkspaceGoalRegion(ss,Tw_0,Te_w,bounds);
```

Customize The State Validator

The customized state validator, `ExampleHelperValidatorRigidBodyTree`, provides rigid body collision checking between the robot and the environment. This validator checks sampled configurations and the planner should discard invalid states.

```
sv = ExampleHelperValidatorRigidBodyTree(ss);
```

```
% Add obstacles in the environment
addFixedObstacle(sv,tabletop1, 'tabletop1', [71 161 214]/256);
addFixedObstacle(sv,tabletop2, 'tabletop2', [71 161 214]/256);
addFixedObstacle(sv,can, 'can', 'r');
addFixedObstacle(sv,floor, 'floor', [1,0.5,0]);

% Skip collision checking for certain bodies for performance
skipCollisionCheck(sv,'root'); % root will never touch any obstacles
skipCollisionCheck(sv,'j2s7s300_link_base'); % base will never touch any obstacles
skipCollisionCheck(sv,'j2s7s300_end_effector'); % this is a virtual frame

% Set the validation distance
sv.ValidationDistance = 0.01;
```

Plan The Grasp Motion

Use the `plannerRRT` object with the customized state space and state validator objects. Specify the start and goal configurations by using `inverseKinematics` to solve for configurations based on the end-effector pose. Specify the `GoalReachedFcn` using `exampleHelperIsStateInWorkspaceGoalRegion`, which checks if a path reaches the goal region.

```
% Set random seeds for repeatable results
rng(0, 'twister') % 0

% Compute the reference goal configuration. Note this is applicable only when goal bias is large
Te_0ref = Tw_0*Te_w; % Reference end-effector pose in world coordinates, derived from WGR
ik = inverseKinematics('RigidBodyTree',kin);
refGoalConfig = ik(ss.EndEffector,Te_0ref,ones(1,6),homeConfiguration(ss.RigidBodyTree));

% Compute the initial configuration (end-effector is initially under the table)
T = Te_0ref;
T(1,4) = 0.3;
T(2,4) = 0.0;
T(3,4) = 0.4;
initConfig = ik(ss.EndEffector,T,ones(1,6),homeConfiguration(ss.RigidBodyTree));

% Create the planner from previously created state space and state validator
planner = plannerRRT(ss,sv);

% If a node in the tree falls in the WGR, a path is considered found.
planner.GoalReachedFcn = @exampleHelperIsStateInWorkspaceGoalRegion;

% Set the max connection distance.
planner.MaxConnectionDistance = 0.5;

% With WGR, there is no need to specify a particular goal configuration (use
% initConfig to hold the place).
% As a result, one can set GoalBias to zero.
planner.GoalBias = 0;
[pthObj,solnInfo] = plan(planner,initConfig, initConfig);
```

Visualize The Grasp Motion

The found path is first smoothed through a recursive corner-cutting strategy [3], before the motion is animated.


```
% Smooth the path.
interpolate(pthObj,100);
newPathObj = exampleHelperPathSmoothing(pthObj,sv);
interpolate(newPathObj,200);

figure
states = newPathObj.States;

% Draw the robot.
ax = show(kin,states(1,:));
zlim(ax, [-0.03, 1.4])
xlim(ax, [-1, 1])
ylim(ax, [-1, 1])

% Render the environment.
hold on
showObstacles(sv, ax);
view(146, 33)
camzoom(1.5)

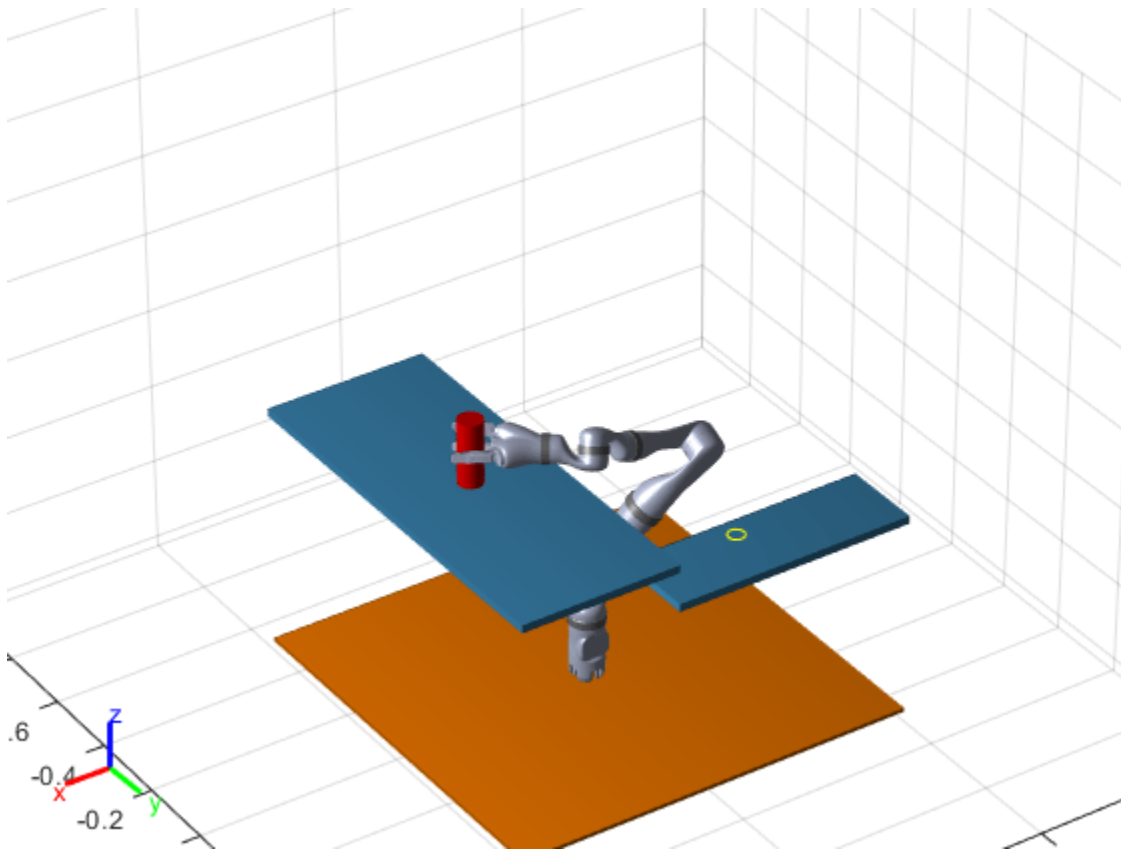
% Show the motion.
for i = 2:length(states)
    show(kin,states(i,:), 'PreservePlot',false, 'Frames', 'off', 'Parent',ax);
    drawnow
end

q = states(i,:);

% Grab the can.
q = exampleHelperEndEffectorGrab(sv, 'can', q, ax);

Set a target psotion for the can on the other tabletop.

targetPos = [-0.15,0.35,0.51];
exampleHelperDrawHorizontalCircle(targetPos,0.02, 'y',ax);
```



Plan The Move Motion

During the move motion, keep the cylinder can level at all time to avoid spill. Specify an additional constraint on the interim manipulator configurations for the RRT planner. Turn the constraint on by setting the UseConstrainedSampling property to true.

```

Tw_0 = trvec2tform(targetPos+[0,0,0.08]);
Te_w = rotm2tform(R);
bounds = [0 0;          % x
          0 0;          % y
          0 0;          % z
          0 0;          % R
          0 0;          % P
          -pi pi];      % Y
setWorkspaceGoalRegion(ss,Tw_0,Te_w,bounds);
ss.UseConstrainedSampling = true;
planner.MaxConnectionDistance = 0.05;
[pthObj2,~] = plan(planner,q,q);

```

Visualize the motion.

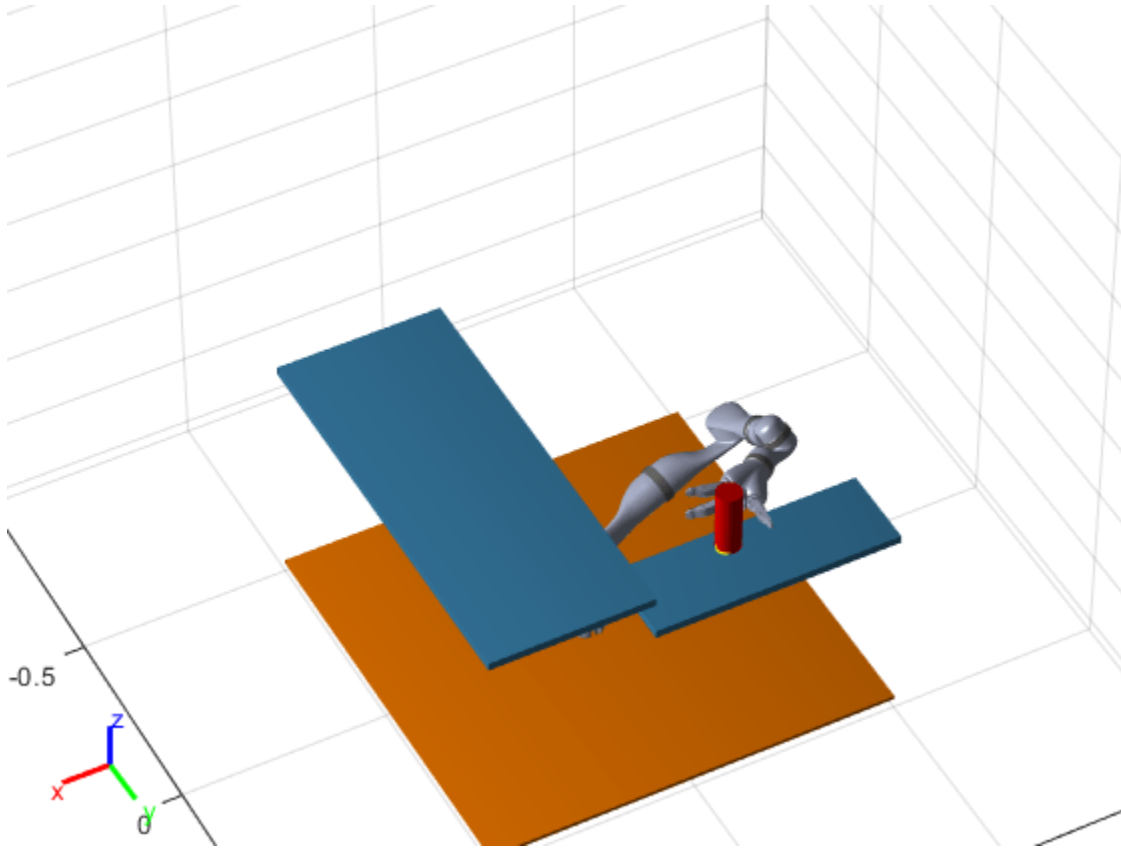
```

states = pthObj2.States;

view(ax, 152,45)
for i = 2:length(states)
    show(kin,states(i,:), 'PreservePlot', false, 'Frames', 'off', 'Parent', ax);
    drawnow
end

```

```
q = states(i,:);  
  
% let go of the can  
q = exampleHelperEndEffectorRelease(sv,q,ax);
```



References

- [1] D. Berenson, S. Srinivasa, D. Ferguson, A. Collet, and J. Kuffner, "Manipulation Planning with Workspace Goal Regions", in *Proceedings of IEEE International Conference on Robotics and Automation*, 2009, pp.1397-1403
- [2] D. Berenson, S. Srinivasa, and J. Kuffner, "Task Space Regions: A Framework for Pose-Constrained Manipulation Planning", *International Journal of Robotics Research*, Vol. 30, No. 12 (2011): 1435-1460
- [3] P. Chen, and Y. Hwang, "SANDROS: A Dynamic Graph Search Algorithm for Motion Planning", *IEEE Transaction on Robotics and Automation*, Vol. 14 No. 3 (1998): 390-403

Dynamic Replanning on an Indoor Map

This example shows how to perform dynamic replanning on a warehouse map with a range finder and an A* path planner.

Create Warehouse Map

Define a `binaryOccupancyMap` that contains the floor plan for the warehouse. This information will be used to create an initial route from the warehouse entrance to the package pickup.

```
map = binaryOccupancyMap(100, 80, 1);
occ = zeros(80, 100);

occ(1,:) = 1;
occ(end,:) = 1;
occ([1:30, 51:80],1) = 1;
occ([1:30, 51:80],end) = 1;

occ(40,20:80) = 1;
occ(28:52,[20:21 32:33 44:45 56:57 68:69 80:81]) = 1;

occ(1:12, [20:21 32:33 44:45 56:57 68:69 80:81]) = 1;

occ(end-12:end, [20:21 32:33 44:45 56:57 68:69 80:81]) = 1;

setOccupancy(map, occ)

figure
show(map)
title('Warehouse Floor Plan')
```



Plan Route to Pickup

Create a `plannerHybridAStar` and use the previously created floor plan to generate an initial route.

```
% Create map that will be updated with sensor readings
estMap = occupancyMap(occupancyMatrix(map));

% Create a map that will inflate the estimate for planning
inflateMap = occupancyMap(estMap);

vMap = validatorOccupancyMap;
vMap.Map = inflateMap;
vMap.ValidationDistance = .1;
planner = plannerHybridAStar(vMap, 'MinTurningRadius', 4);

entrance = [1 40 0];
packagePickupLocation = [63 44 -pi/2];
route = plan(planner, entrance, packagePickupLocation);
route = route.States;

% Get poses from the route.
rsConn = reedsSheppConnection('MinTurningRadius', planner.MinTurningRadius);
startPoses = route(1:end-1,:);
endPoses = route(2:end,:);

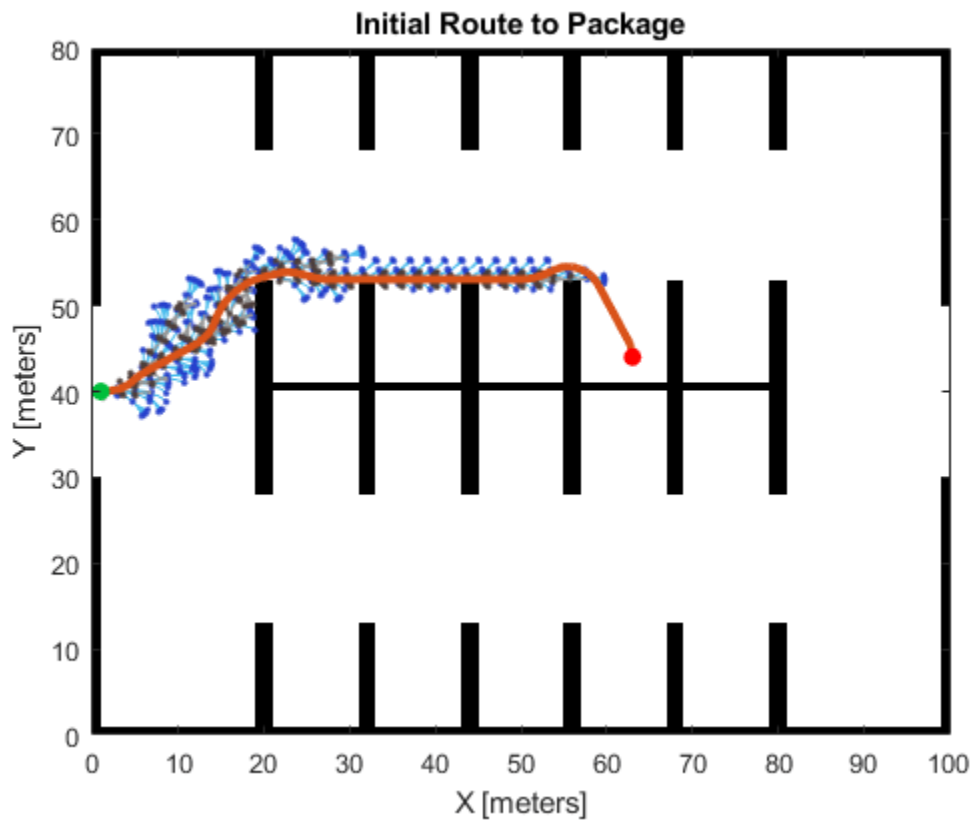
rsPathSegs = connect(rsConn, startPoses, endPoses);
```

```

poses = [];
for i = 1:numel(rsPathSegs)
    lengths = 0:0.1:rsPathSegs{i}.Length;
    [pose, ~] = interpolate(rsPathSegs{i}, lengths);
    poses = [poses; pose];
end

figure
show(planner)
title('Initial Route to Package')

```



Place an Obstacle in the Route

Add an obstacle to the map that is on the route the forklift will take to the package.

```

obstacleWidth = 6;
obstacleHeight = 11;
obstacleBottomLeftLocation = [34 49];
values = ones(obstacleHeight, obstacleWidth);
setOccupancy(map, obstacleBottomLeftLocation, values)

figure
show(map)
title('Warehouse Floor Plan with Obstacle')

```



Specify the Range Finder

Create the range finder using the `rangeSensor` object.

```
rangefinder = rangeSensor('HorizontalAngle', pi/3);
numReadings = rangefinder.NumReadings;
```

Update the Route Based on Range Finder Readings

Advance the forklift forward using the poses from the path planner. Get the new obstacle detections from the range finder and insert them into the estimated map. If the route is now occupied in the updated map, recalculate the route. Repeat until the goal is reached.

```
% Setup visualization.
helperViz = HelperUtils;
figure
show(inflateMap);
hold on
robotPatch = helperViz.plotRobot(gca, poses(1,:));
rangesLine = helperViz.plotScan(gca, poses(1,:), ...
    NaN(numReadings,1), ones(numReadings,1));
axesColors = get(gca, 'ColorOrder');
routeLine = helperViz.plotRoute(gca, route, axesColors(2,:));
traveledLine = plot(gca, NaN, NaN);
title('Forklift Route to Package')
hold off

idx = 1;
```

```
tic;
while idx <= size(poses,1)
    % Insert range finder readings into estimated map.
    [ranges, angles] = rangefinder(poses(idx,:), map);
    insertRay(estMap, poses(idx,:), ranges, angles, ...
        rangefinder.Range(end));

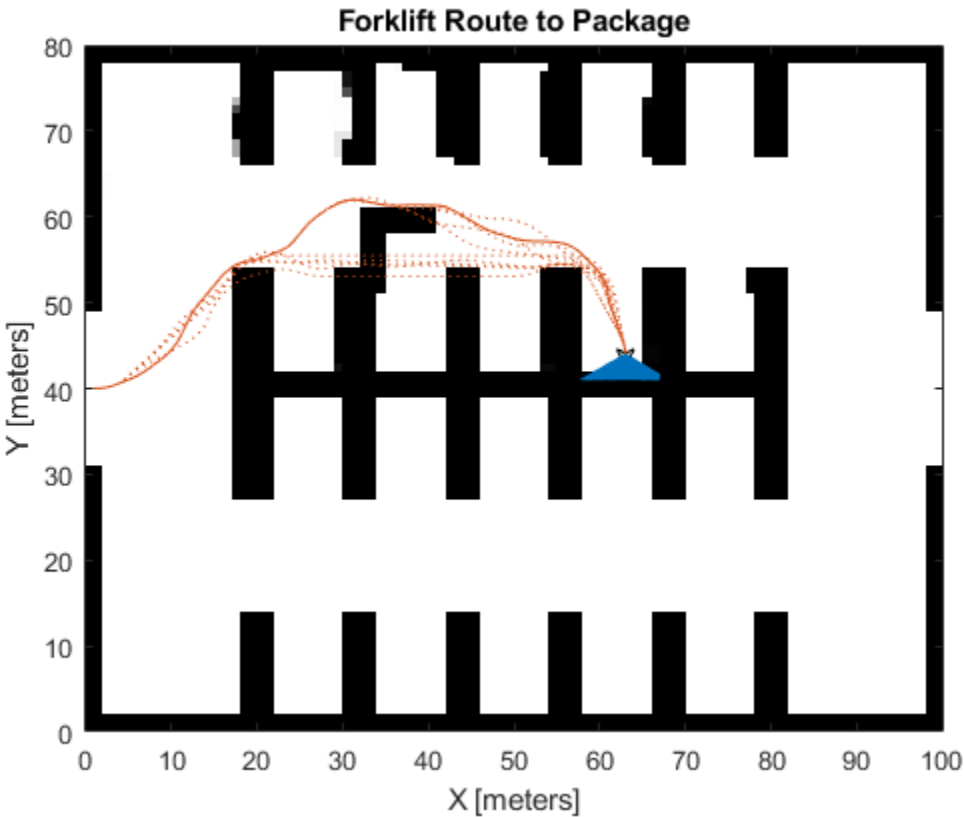
    % Reset and reinflate planning map
    setOccupancy(inflateMap, getOccupancy(estMap));
    inflate(inflateMap,1,'grid');

    % Update visualization.
    show(inflateMap, 'FastUpdate', true);
    helperViz.updateWorldMap(robotPatch, rangesLine, traveledLine, ...
        poses(idx,:), ranges, angles)
    drawnow

    % Regenerate route when obstacles are detected in the current one.
    isRouteOccupied = any(checkOccupancy(inflateMap, poses(:,1:2)));
    if isRouteOccupied && (toc > 0.1)
        % Calculate new route.
        route = plan(planner, poses(idx,:), packagePickupLocation);
        route = route.States;

        % Get poses from the route.
        startPoses = route(1:end-1,:);
        endPoses = route(2:end,:);
        rsPathSegs = connect(rsConn, startPoses, endPoses);
        poses = [];
        for i = 1:numel(rsPathSegs)
            lengths = 0:0.1:rsPathSegs{i}.Length;
            [pose, ~] = interpolate(rsPathSegs{i}, lengths);
            poses = [poses; pose]; %#ok<AGROW>
        end

        routeLine = helperViz.updateRoute(routeLine, route);
        idx = 1;
        tic;
    else
        idx = idx + 1;
    end
end
```

Highway Lane Change

This example shows how to perceive surround-view information and use it to design an automated lane change maneuver system for highway driving scenarios.

Introduction

An automated lane change maneuver (LCM) system enables the ego vehicle to automatically move from one lane to another lane. An LCM system models the longitudinal and lateral control dynamics for an automated lane change. LCM systems scan the environment for most important objects (MIOs) using onboard sensors, identify an optimal trajectory that avoids these objects, and steer the ego vehicle along the identified trajectory.

This example shows how to create a test bench model to test the sensor fusion, planner, and controller components of an LCM system. This example uses five vision sensors and one radar sensor to detect other vehicles from the surrounding view of the ego vehicle. It uses a joint probabilistic data association (JPDA) based tracker to track the fused detections from these multiple sensors. The lane change planner then generates a feasible trajectory for the tracks to negotiate a lane change that is executed by the lane change controller. In this example, you:

- **Partition the algorithm and test bench** — The model is partitioned into lane change algorithm models and a test bench model. The algorithm models implement the individual components of the LCM system. The test bench includes the integration of the algorithm models and testing framework.
- **Explore the test bench model** — The test bench model contains the testing framework, which includes the sensors and environment, ego vehicle dynamics model, and metrics assessment using ground truth.
- **Explore the algorithm models** — Algorithm models are reference models that implement the sensor fusion, planner, and controller components to build the lane change application.
- **Simulate and visualize system behavior** — Simulate the test bench model to test the integration of sensor fusion and tracking with planning and controls to perform lane change maneuvers on a curved road with multiple vehicles.
- **Explore other scenarios** — These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own LCM system.

Partition Algorithm and Test Bench

The model is partitioned into separate algorithm models and a test bench model.

- **Algorithm models** — Algorithm models are reference models that implement the functionality of individual components.
- **Test bench model** — The **Highway Lane Change Test Bench** specifies the stimulus and environment for testing the algorithm models.

Explore Test Bench Model

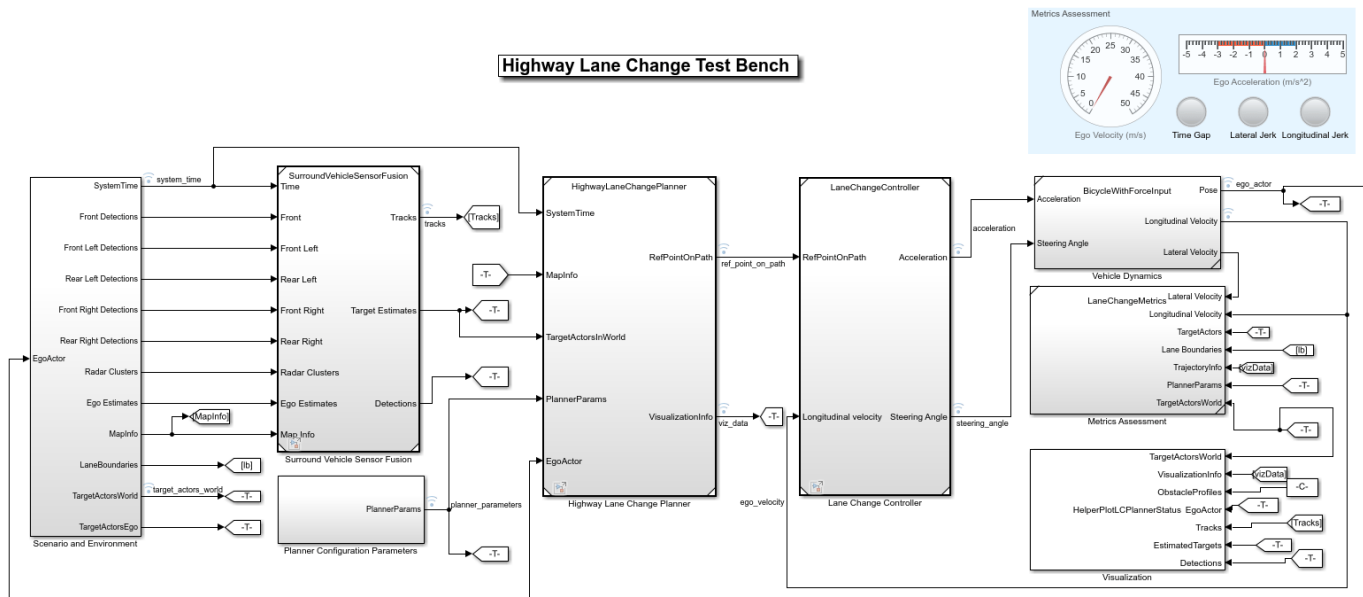
In this example, you use a system-level simulation test bench model to explore the behavior of a probabilistic sensor-based LCM system.

To explore the test bench model, open a working copy of the project example files. MATLAB® copies the files to an example folder so you can edit them.

```
addpath(fullfile(matlabroot,"toolbox","driving","drivingdemos"));
helperDrivingProjectSetup("HighwayLaneChange.zip",workDir=pwd);
```

Open the system-level simulation test bench model.

```
open_system("HighwayLaneChangeTestBench")
```



Opening this model runs the `helperSLHighwayLaneChangeSetup` function, which initializes the road scenario using the `drivingScenario` (Automated Driving Toolbox) object in the base workspace. It also configures the sensor configuration parameters, tracker design parameters, planner configuration parameters, controller design parameters, vehicle model parameters, and the Simulink® bus signals required for defining the inputs and outputs for the `HighwayLaneChangeTestBench` model.

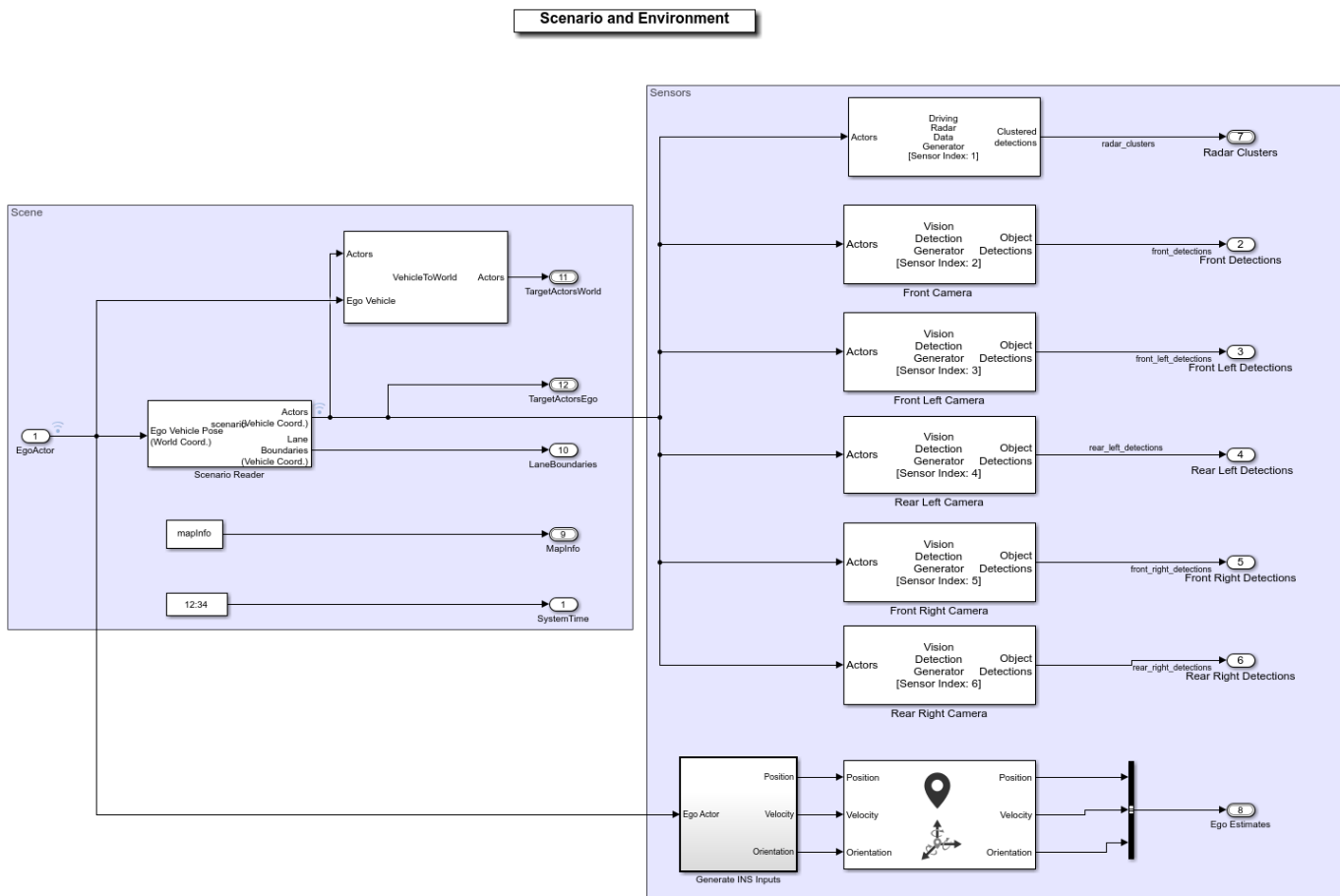
The test bench model contains these subsystems:

- **Scenario and Environment** — Subsystem that specifies the scene, vehicles, sensors, and map data used for simulation. This example uses five vision sensors, one radar sensor, and an INS sensor.
- **Surround Vehicle Sensor Fusion** — Subsystem that fuses the detections from multiple sensors to produce tracks.
- **Planner Configuration Parameters** — Subsystem that specifies the configuration parameters required for the planner algorithm.
- **Highway Lane Change Planner** — Subsystem that implements the lane change planner algorithm for highway driving.
- **Lane Change Controller** — Subsystem that specifies the path-following controller that generates control commands to steer the ego vehicle along the generated trajectory.
- **Vehicle Dynamics** — Subsystem that specifies the dynamic model for the ego vehicle.
- **Metrics Assessment** — Subsystem that specifies metrics to assess system-level behavior.

The Highway Lane Change Planner, Lane Change Controller, and Metrics Assessment subsystems are the same as those in the “Highway Lane Change Planner and Controller” (Automated Driving Toolbox) example. However, whereas the lane change planner in the Highway Lane Change Planner and Controller example, uses ground truth information from the scenario to detect MIOs, the lane change planner in this example uses tracks from surround vehicle sensor fusion to detect the MIOs. The Vehicle Dynamics subsystem models the ego vehicle using a Bicycle Model block, and updates its state using commands received from the Lane Change Controller subsystem.

The Scenario and Environment subsystem uses the Scenario Reader (Automated Driving Toolbox) block to provide road network and vehicle ground truth positions. This block also outputs map data required for the highway lane change planner algorithm. This subsystem outputs the detections from the vision sensors, clusters from the radar sensor, and ego-estimated position from the INS sensor required for the sensor fusion and tracking algorithm. Open the Scenario and Environment subsystem.

open_system("HighwayLaneChangeTestBench/Scenario and Environment")

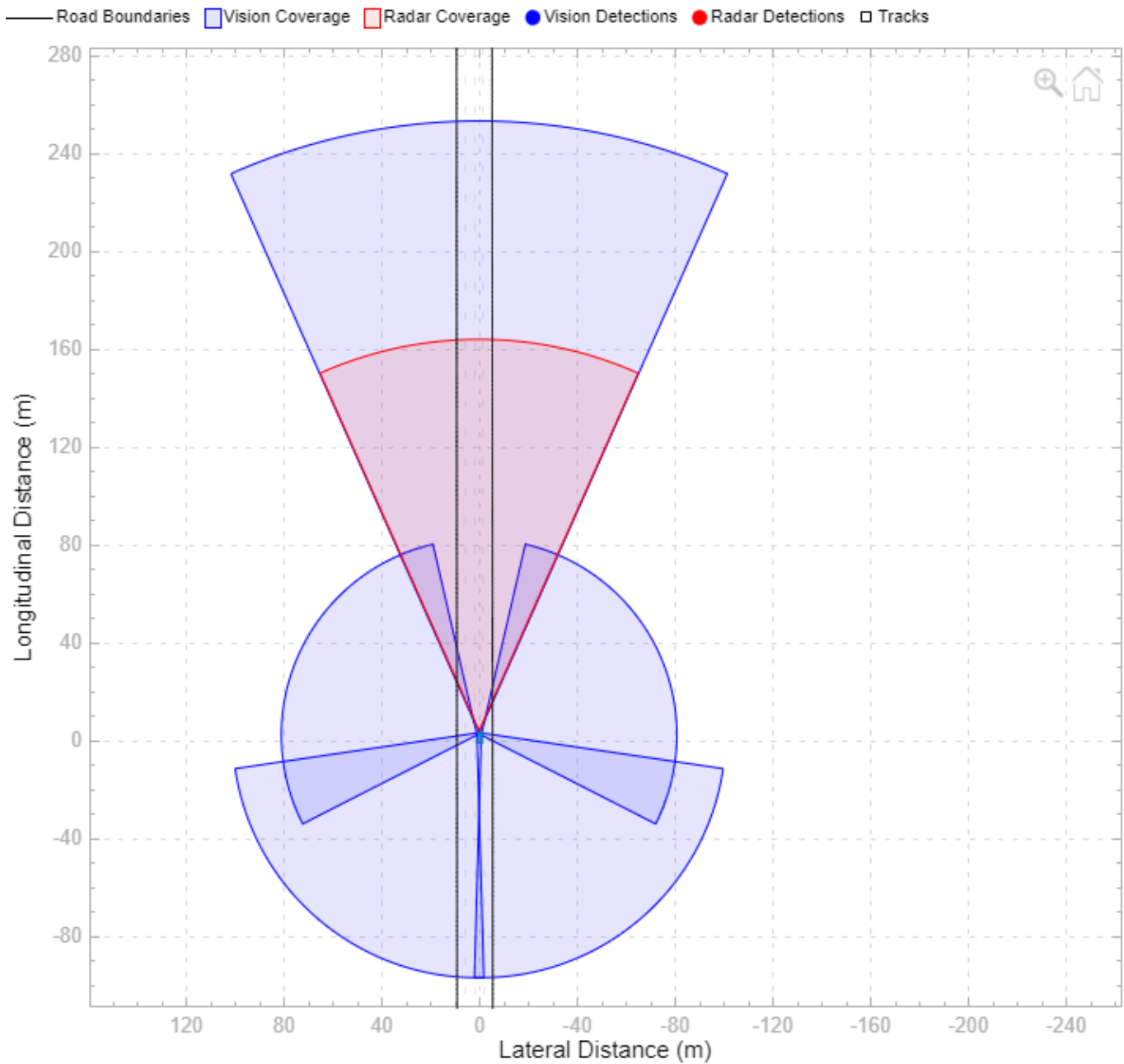


- The Scenario Reader (Automated Driving Toolbox) block configures the driving scenario and outputs actor poses, which control the positions of the target vehicles.
- The Vehicle To World (Automated Driving Toolbox) block converts actor poses from the coordinates of the ego vehicle to the world coordinates.

- The Vision Detection Generator (Automated Driving Toolbox) block simulates object detections using a camera sensor model.
- The Driving Radar Data Generator (Automated Driving Toolbox) block simulates object detections based on a statistical model. It also outputs clustered object detections for further processing.
- The INS (Automated Driving Toolbox) block models the measurements from the inertial navigation system and global navigation satellite system and outputs the fused measurements. It outputs the noise-corrupted position, velocity, and orientation of the ego vehicle.

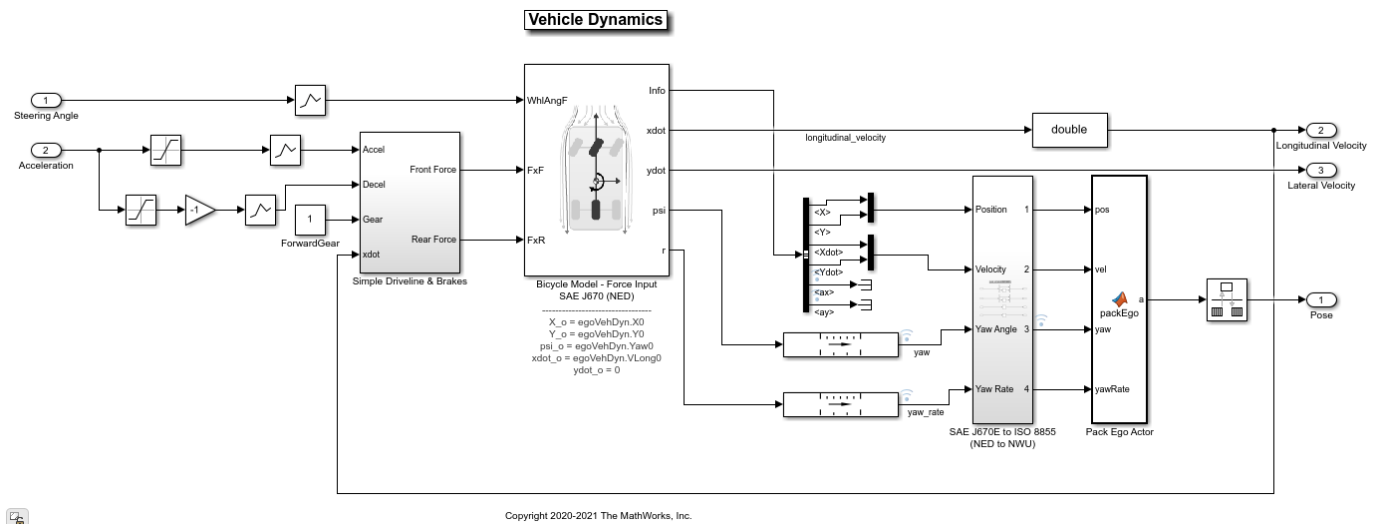
The subsystem configures five vision sensors and a radar sensor to capture the surround view of the vehicle. These sensors are mounted on different locations on the ego vehicle to capture a 360-degree view.

The **Bird's-Eye Scope** displays sensor coverage using a cuboid representation. The radar coverage area and detections are in red. The vision coverage area and detections are in blue.



The `Vehicle Dynamics` subsystem uses a `Bicycle Model` block to model the ego vehicle. For more details on the `Vehicle Dynamics` subsystem, see the “Highway Lane Following” (Automated Driving Toolbox) example. Open the `Vehicle Dynamics` subsystem.

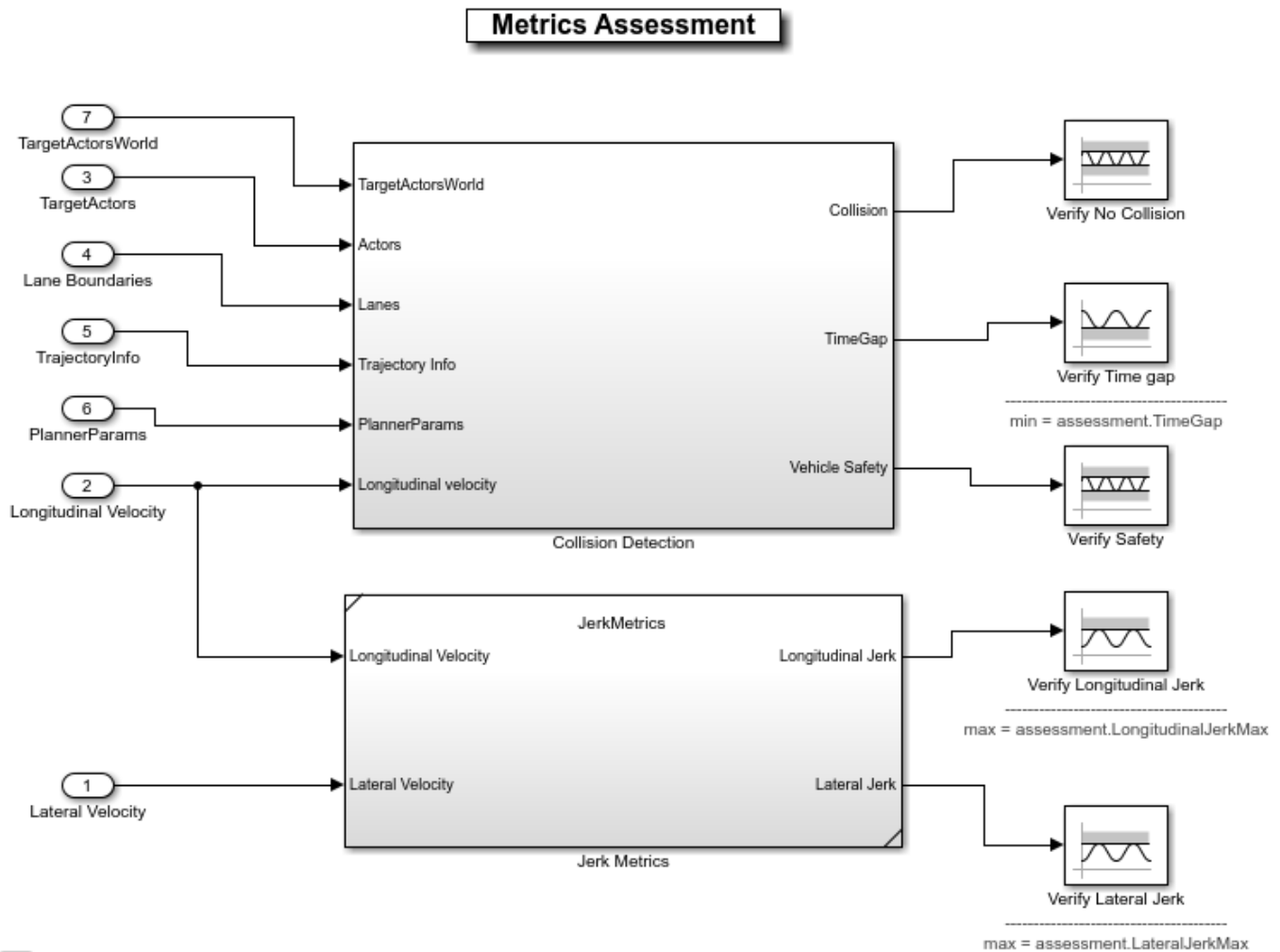
```
open_system("HighwayLaneChangeTestBench/Vehicle Dynamics");
```



The **Bicycle Model** block implements a rigid two-axle single-track vehicle body model to calculate longitudinal, lateral, and yaw motion. The block accounts for body mass, aerodynamic drag, and weight distribution between the axles due to acceleration and steering. For more details, see **Bicycle Model (Automated Driving Toolbox)** (Automated Driving Toolbox).

The **Metric Assessment** subsystem enables system-level metric evaluations using the ground truth information from the scenario. Open the **Metrics Assessment** subsystem.

```
open_system("HighwayLaneChangeTestBench/Metrics Assessment")
```



- The **Collision Detection** subsystem detects the collision of the ego vehicle with other vehicles and halts the simulation if it detects a collision. The subsystem also computes the **TimeGap** parameter using the distance to the lead vehicle (headway) and the longitudinal velocity of the ego vehicle. This parameter is evaluated against prescribed limits.
- The **Jerk Metrics** subsystem computes the **LongitudinalJerk** and **LateralJerk** parameters using longitudinal velocity and lateral velocity, respectively. These parameters are evaluated against prescribed limits.

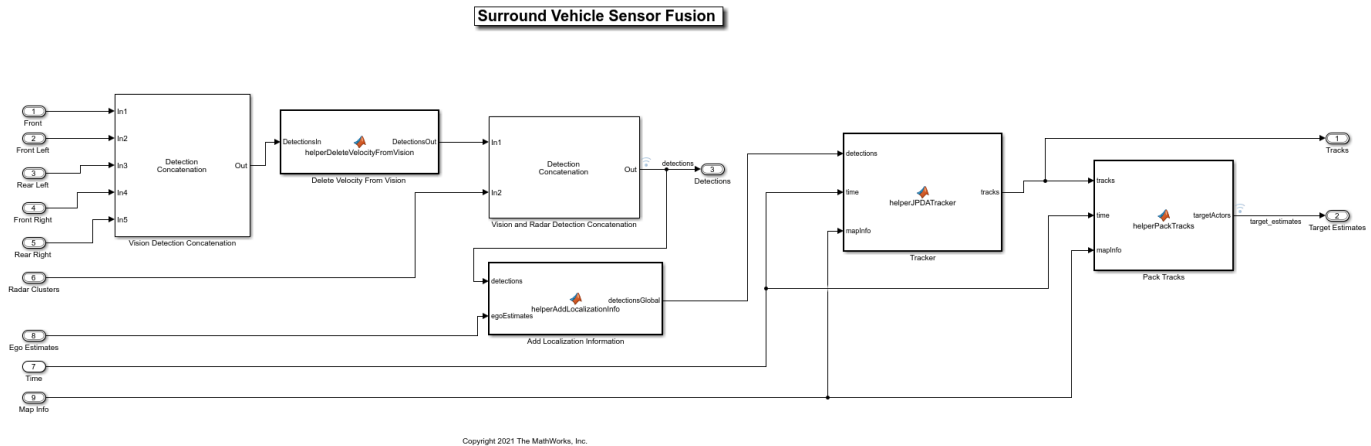
For more details on how to validate the metrics automatically using Simulink Test, see the “Automate Testing for Highway Lane Change” (Automated Driving Toolbox) example.

Explore Algorithm Models

The lane change system is developed by integrating the surround vehicle sensor fusion, lane-change planner, and lane-following controller components.

The surround vehicle sensor fusion algorithm model fuses vehicle detections from cameras and radar sensors and tracks the detected vehicles using the central-level tracking method. Open the Surround Vehicle Sensor Fusion algorithm model.

```
open_system("SurroundVehicleSensorFusion")
```



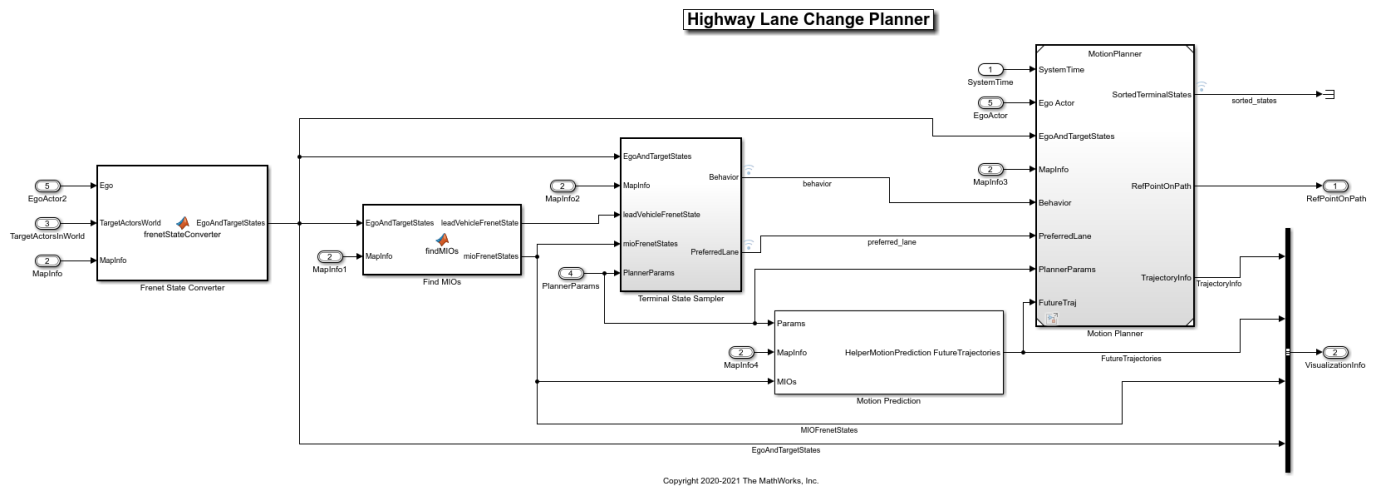
The surround vehicle sensor fusion model takes the vehicle detections from vision sensors and clusters from the radar sensor as inputs.

- The **Vision Detection Concatenation** block concatenates the vision detections.
- The **Delete Velocity From Vision** block is a MATLAB Function block that deletes velocity information from vision detections.
- The **Vision and Radar Detection Concatenation** block concatenates the vision and radar detections.
- The **Add Localization Information** block is a MATLAB Function block that adds localization information for the ego vehicle to the concatenated detections using an estimated ego vehicle pose from the INS sensor. This enables the tracker to track in the global frame, and minimizes the effect on the tracks of lane change maneuvers by the ego vehicle.
- The **helperJPDATracker** block performs fusion and manages the tracks of stationary and moving objects. The tracker fuses the information contained in the concatenated detections and tracks the objects around the ego vehicle. It estimates tracks in the Frenet coordinate system. It uses **mapInfo** from the scenario to estimate the tracks in Frenet coordinate system. The tracker then outputs a list of confirmed tracks. These tracks are updated at a prediction time driven by a digital clock in the **Scenario** and **Environment** subsystem.

For more details on the algorithm, see the “Object Tracking and Motion Planning Using Frenet Reference Path” (Automated Driving Toolbox) example.

The highway lane change planner is a fundamental component of a highway lane change system. This component is expected to handle different driving behaviors to safely navigate the ego vehicle from one point to another point. The **Highway Lane Change Planner** algorithm model contains a terminal state sampler, motion planner, and motion prediction module. The terminal state sampler samples terminal states based on the planner parameters and the current state of both the ego vehicle and other vehicles in the scenario. The motion prediction module predicts the future motion of MIOs. The motion planner samples trajectories and outputs an optimal trajectory. Open the **Highway Lane Change Planner** algorithm model.

```
open_system("HighwayLaneChangePlanner")
```

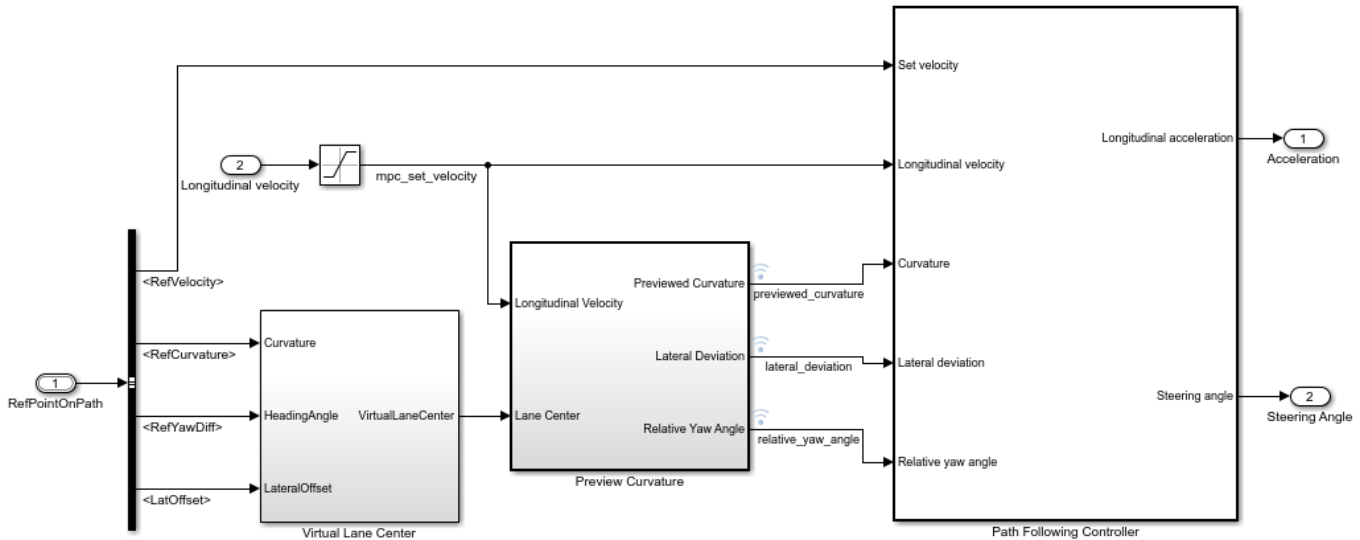


The algorithm model implements the main algorithm for the highway lane change system. The reference model reads map data, actor poses (in world coordinates), and planner parameters from the Scenario and Environment subsystem to perform trajectory planning. The model uses the Frenet coordinate system to find the MIOs surrounding the ego vehicle. Then, the model samples terminal states for different behaviors, predicts the motion of target actors, and generates multiple trajectories. Finally, the model evaluates the costs of generated trajectories and checks for the possibility of collision and kinematic feasibility to estimate the optimal trajectory. For more details, see the “Generate Code for Highway Lane Change Planner” (Automated Driving Toolbox) example.

The Lane Change Controller reference model simulates a path-following control mechanism that keeps the ego vehicle traveling along the generated trajectory while tracking a set velocity. Open the Lane Change Controller reference model.

```
open_system("LaneChangeController");
```

Lane Change Controller



Copyright 2019-2021 The MathWorks, Inc.

The controller adjusts both the longitudinal acceleration and front steering angle of the ego vehicle to ensure that the ego vehicle travels along the generated trajectory. The controller computes optimal control actions while satisfying velocity, acceleration, and steering angle constraints using adaptive model predictive control (MPC). For more details on the integration of the highway lane change planner and controller, see the “Highway Lane Change Planner and Controller” (Automated Driving Toolbox) example.

Simulate and Visualize System Behavior

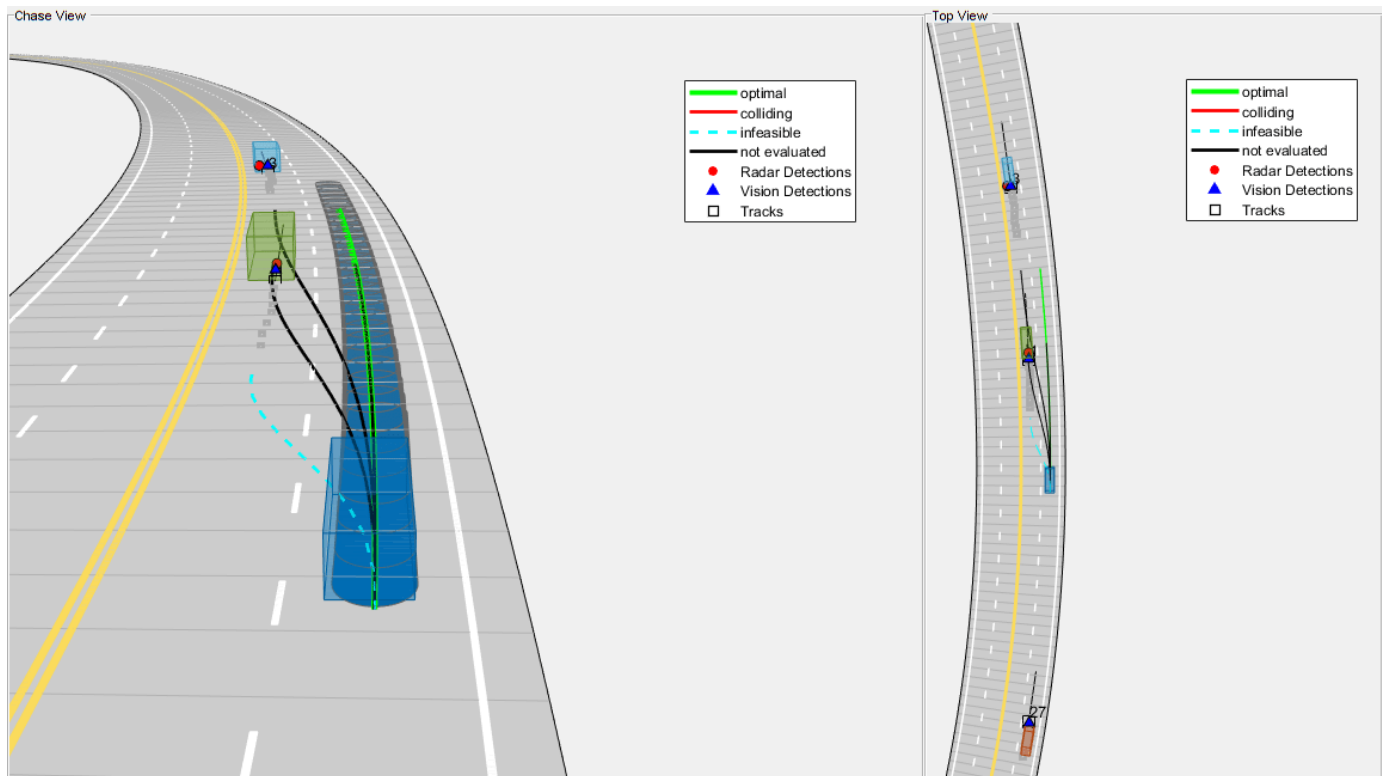
Set up and run the `HighwayLaneChangeTestBench` simulation model to visualize the behavior of the system during a lane change. The `Visualization` block in the model creates a MATLAB figure that shows the chase view and top view of the scenario and plots the ego vehicle, tracks, sampled trajectories, capsule list, and other vehicles in the scenario.

Disable the MPC update messages.

```
mpcverbosity("off");
```

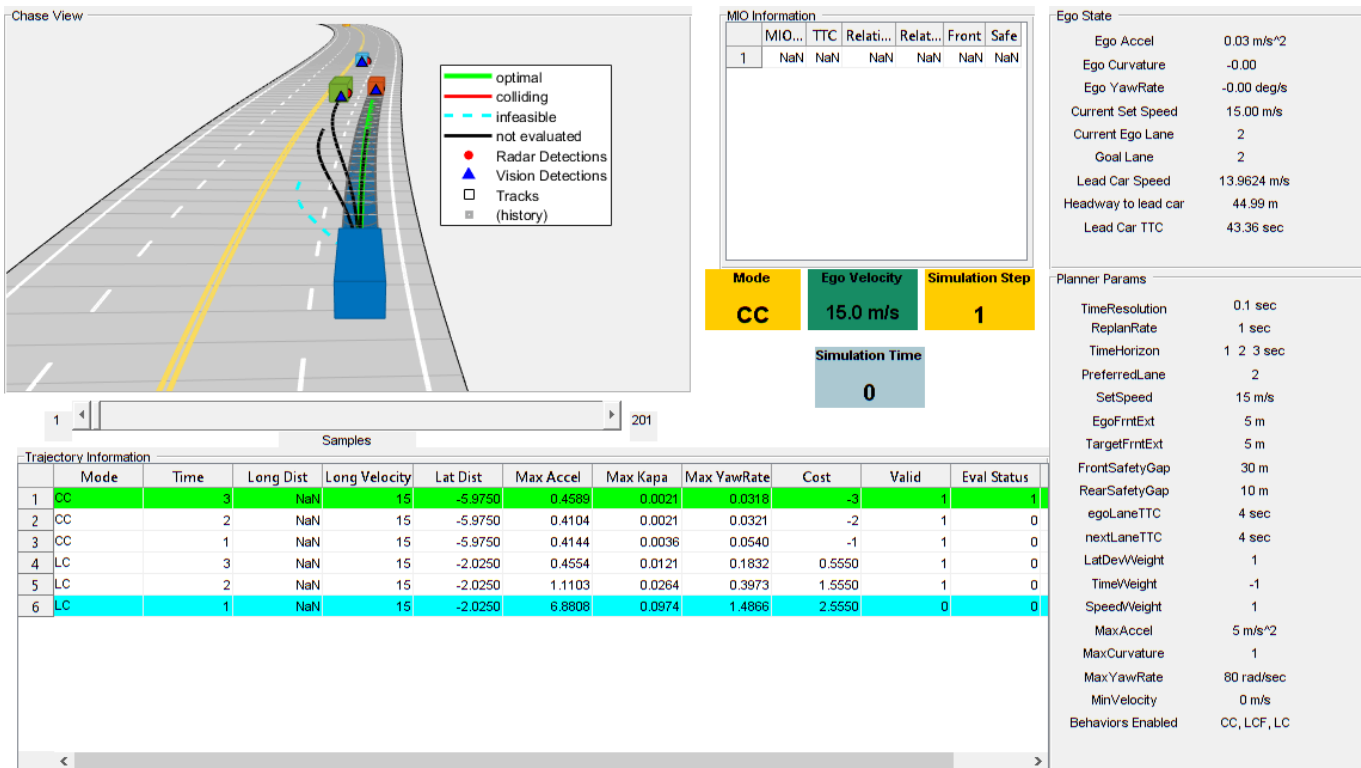
Configure the `HighwayLaneChangeTestBench` model to use the `scenario_LC_15_StopnGo_Curved` scenario.

```
helperSLHighwayLaneChangeSetup(scenarioFcnName="scenario_LC_15_StopnGo_Curved");
sim("HighwayLaneChangeTestBench");
```



During the simulation, the model logs signals to the base workspace as `logout`. You can analyze the simulation results and debug any failures in the system behavior using the `helperAnalyzeLCSimulationResults` function. The function creates a MATLAB figure and plots a chase view of the scenario. For more details on this figure, see the “Highway Lane Change Planner and Controller” (Automated Driving Toolbox) to example. Run the function and explore the plot.

```
helperAnalyzeLCSimulationResults(logout);
```



Explore Other Scenarios

In this example, you have explored the system behavior for the scenario `scenario_LC_15_StopnGo_Curved` scenario, but you can use the same test bench model to explore other scenarios. This is a list of scenarios that are compatible with the `HighwayLaneChangeTestBench` model.

```

scenario_LC_01_SlowMoving
scenario_LC_02_SlowMovingWithPassingCar
scenario_LC_03_DisabledCar
scenario_LC_04_CutInWithBrake
scenario_LC_05_SingleLaneChange
scenario_LC_06_DoubleLaneChange
scenario_LC_07_RightLaneChange
scenario_LC_08_SlowmovingCar_Curved
scenario_LC_09_CutInWithBrake_Curved
scenario_LC_10_SingleLaneChange_Curved
scenario_LC_11_MergingCar_HighwayEntry
scenario_LC_12_CutInCar_HighwayEntry
scenario_LC_13_DisabledCar_Ushape
scenario_LC_14_DoubleLaneChange_Ushape
scenario_LC_15_StopnGo_Curved [Default]

```

Each of these scenarios have been created using the Driving Scenario Designer (Automated Driving Toolbox) and exported to a scenario file. Examine the comments in each file for more details on the road and vehicles in each scenario. You can configure the `HighwayLaneChangeTestBench` model and workspace to simulate these scenarios using the `helperSLHighwayLaneChangeSetup` function. For example, you can configure the simulation for a curved road scenario using this command.

```
helperSLHighwayLaneChangeSetup(scenarioFcnName="scenario_LC_10_SingleLaneChange_Curved");
```

Conclusion

In this example, you designed and simulated a highway lane change maneuver system using information perceived from surround view. This example showed how to integrate sensor fusion, planner, and controller components to simulate a highway lane change system in a closed-loop environment. The example also demonstrated various evaluation metrics to validate the performance of the designed system. If you have a Simulink Coder™ license and Embedded Coder™ license, you can generate ready-to-deploy code of the algorithm models for an embedded real-time target (ERT).

Enable the MPC update messages again.

```
mpcverbosity("on");
```

Path Following with Obstacle Avoidance in Simulink®

This example shows you how to use Simulink to avoid obstacles while following a path for a differential drive robot. This example uses ROS to send and receive information from a MATLAB®-based simulator. You can replace the simulator with other ROS-based simulators such as Gazebo®.

Prerequisites: “Connect to a ROS-enabled Robot from Simulink®” (ROS Toolbox)

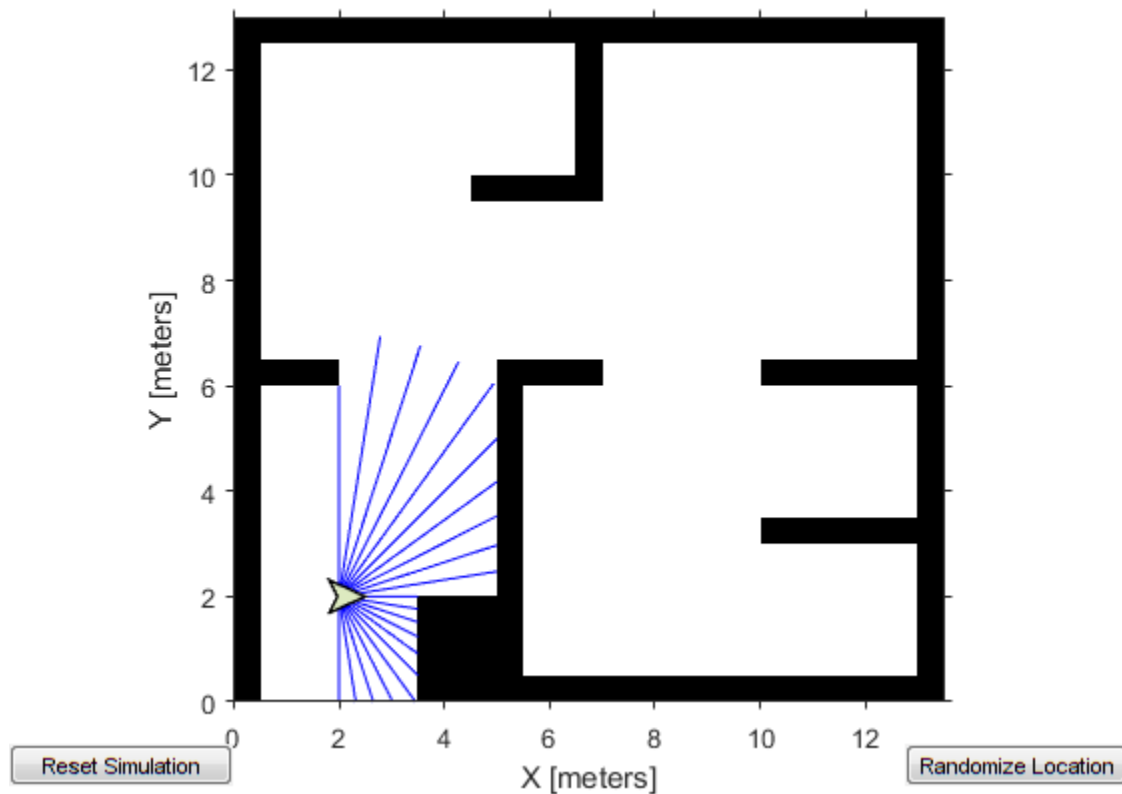
Introduction

This example uses a model that implements a path following controller with obstacle avoidance. The controller receives the robot pose and laser scan data from the simulated robot and sends velocity commands to drive the robot on a given path. You can adjust parameters while the model is running and observe the effect on the simulated robot.

Start a Robot Simulator

Start a simple MATLAB-based simulator:

- Type `roslaunch` (ROS Toolbox) at the MATLAB command line. This creates a local ROS master with network address (URI) of `http://localhost:11311`.
- Type `ExampleHelperSimulinkRobotROS('ObstacleAvoidance')` to start the Robot Simulator. This opens a figure window:



This MATLAB-based simulator is a ROS-based simulator for a differential-drive robot. The simulator receives and sends messages on the following topics:

- It receives velocity commands, as messages of type `geometry_msgs/Twist`, on the `/mobile_base/commands/velocity` topic
- It sends ground truth robot pose information, as messages of type `nav_msgs/Odometry`, to the `/ground_truth_pose` topic
- It sends laser range data, as messages of type `sensor_msgs/LaserScan`, to the `/scan` topic

Replacing the MATLAB-based simulator with Gazebo:

You can also use Gazebo simulator with a simulated TurtleBot®. See “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox) for instructions on setting up the Gazebo environment. See “Connect to a ROS-enabled Robot from Simulink®” (ROS Toolbox) for instructions on setting up network connection with Gazebo. After starting the virtual machine, launch **Gazebo Office** world using the desktop shortcut. The simulated Turtlebot in the Gazebo simulator, receives velocity commands, as messages of type `geometry_msgs/Twist`, on the `/cmd_vel` topic. You also need a localization algorithm to get robot position in the Gazebo. See “Localize TurtleBot Using Monte Carlo Localization” on page 1-248 for instructions on finding robot location in Gazebo environment.

Open Existing Model

This model implements the path following with obstacle avoidance algorithm. The model is divided into four subsystems. The following sections explain each subsystem.

```
open_system('pathFollowingWithObstacleAvoidanceExample');
```

Process Inputs

The 'Inputs' subsystem processes all the inputs to the algorithm.

There are two subscribers to receive data from the simulator. The first subscriber receives messages sent on the `/scan` topic. The laser scan message is then processed to extract scan ranges and angles. The second subscriber receives messages sent on the `/ground_truth_pose` topic. The (x, y) location and Yaw orientation of the robot is then extracted from the pose message.

The path is specified as a set of waypoints. This example uses a 3x2 constant input. You can specify any number of waypoints as an Nx2 array. To change the size of the path at run-time, you can either use a variable sized signal or use a fixed size signal with NaN padding. This example uses a fixed size input with NaN padding for the waypoints that are unknown.

```
open_system('pathFollowingWithObstacleAvoidanceExample/Inputs', 'tab');
```

Compute Velocity and Heading for Path Following

The 'Compute Velocity and Heading for Path Following' subsystem computes the linear and angular velocity commands and the target moving direction using the **Pure Pursuit** block. The Pure Pursuit block is located in the **Mobile Robot Algorithms** sub-library within the **Robotics System Toolbox** tab in the Library Browser. Alternatively, you can type `robotalgslib` on the command-line to open the **Mobile Robot Algorithms** sub-library.

You also need to stop the robot once it reaches a goal point. In this example, the goal is the last waypoint on the path. This subsystem also compares the current robot pose and the goal point to determine if the robot is close to the goal.

```
open_system('pathFollowingWithObstacleAvoidanceExample/Compute Velocity and Heading for Path fol
```

Adjust Velocities to Avoid Obstacles

The 'Adjust Velocities to Avoid Obstacles' subsystem computes adjustments to the linear and angular velocities computed by the path follower.

The Vector Field Histogram block uses the laser range readings to check if the target direction computed using the Pure Pursuit block is obstacle-free or not based on the laser scan data. If there are obstacles along the target direction, the Vector Field Histogram block computes a steering direction that is closest to the target direction and is obstacle-free. The Vector Field Histogram block is also located in the **Mobile Robot Algorithms** sub-library.

The steering direction is NaN value when there are no obstacle-free directions in the sensor field of view. In this case, a recovery motion is required, where the robot turns on-the-spot until an obstacle-free direction is available.

Based on the steering direction, this subsystem computes adjustments in linear and angular velocities.

```
open_system('pathFollowingWithObstacleAvoidanceExample/Adjust Velocities to Avoid Obstacles','tab');
```

Send Velocity Commands

The 'Outputs' subsystem publishes the linear and angular velocities to drive the simulated robot. It adds the velocities computed using the Pure Pursuit path following algorithm with the adjustments computed using the Vector Field Histogram obstacle avoidance algorithm. The final velocities are set on the `geometry_msgs/Twist` message and published on the topic `/mobile_base/commands/velocity`.

This is an enabled subsystem which is triggered when new laser message is received. This means a velocity command is published only when a new sensor information is available. This prevents the robot from hitting the obstacles in case of delay in receiving sensor information.

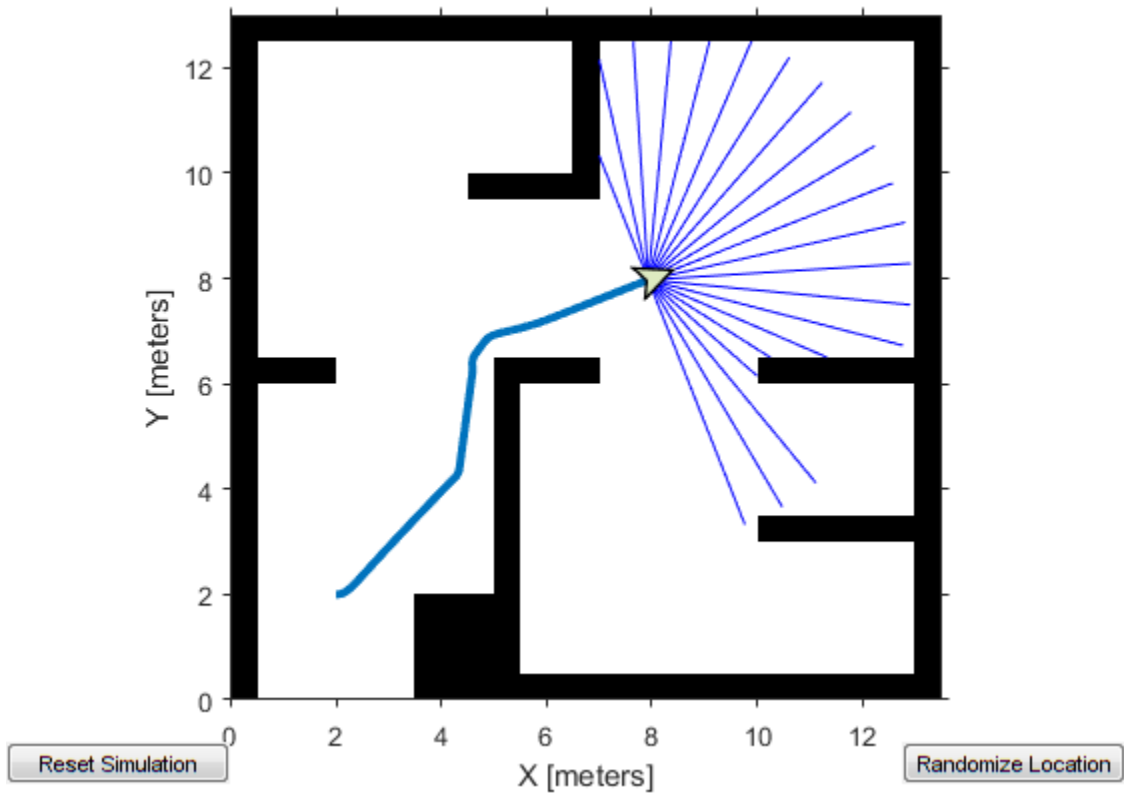
```
open_system('pathFollowingWithObstacleAvoidanceExample/Outputs','tab');
```

Note: To use Gazebo simulator, select the `/cmd_vel` topic in the Publish block.

Configure and Run the Model

Configure and run your model and observe the motion of the robot in the simulator.

- Set simulation Stop time to Inf.
- Click the Play button to start simulation. Observe that the robot starts moving in the simulation.
- While the simulation is running, open 'Compute Velocity and Heading for Path Following' subsystem and double-click on the **Pure Pursuit** block. Change the desired linear velocity parameter to 0.5. Observe increase in the velocity of the robot.
- The default path [2 2; 8 8] passes through an obstacle. Observe that the robot takes a detour around the obstacle to reach the end point of the path.
- Open the 'Inputs' subsystem and double-click on the **Waypoints Input** block. Change the constant value from [2 2;8 8;NaN NaN] to [2 2; 8 8; 12 5]. Notice that robot continues to follow the new path and reaches the new goal point (12,5) while avoiding obstacles.
- To stop the simulation, click the Stop button.



See Also

- “Generate a Standalone ROS Node from Simulink®” (ROS Toolbox)
- “Path Planning in Environments of Different Complexity” (Robotics System Toolbox)

Obstacle Avoidance with TurtleBot and VFH

This example shows how to use ROS Toolbox and a TurtleBot® with vector field histograms (VFH) to perform obstacle avoidance when driving a robot in an environment. The robot wanders by driving forward until obstacles get in the way. The `controllerVFH` object computes steering directions to avoid objects while trying to drive forward.

Optional: If you do not already have a TurtleBot (simulated or real) set up, install a virtual machine with the Gazebo simulator and TurtleBot package. See “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox) to install and set up a TurtleBot in Gazebo.

Connect to the TurtleBot using the IP address obtained from setup.

```
rosinit('192.168.233.133', 11311)
```

```
Initializing global node /matlab_global_node_90736 with NodeURI http://192.168.233.1:61063/
```

Create a publisher and subscriber to share information with the VFH class. The subscriber receives the laser scan data from the robot. The publisher sends velocity commands to the robot.

The topics used are for the simulated TurtleBot. Adjust the topic names for your specific robot.

```
laserSub = rossubscriber('/scan');
[velPub, velMsg] = rospublisher('/mobile_base/commands/velocity');
```

Set up VFH object for obstacle avoidance. Set the `UseLidarScan` property to `true`. Specify algorithm properties for robot specifications. Set target direction to `0` in order to drive straight.

```
vfh = controllerVFH;
vfh.UseLidarScan = true;
vfh.DistanceLimits = [0.05 1];
vfh.RobotRadius = 0.1;
vfh.MinTurningRadius = 0.2;
vfh.SafetyDistance = 0.1;
```

```
targetDir = 0;
```

Set up a Rate object using `rateControl`, which can track the timing of your loop. This object can be used to control the rate the loop operates as well.

```
rate = rateControl(10);
```

Create a loop that collects data, calculates steering direction, and drives the robot. Set a loop time of 30 seconds.

Use the ROS subscriber to collect laser scan data. Create a `lidarScan` object by specifying the ranges and angles. Calculate the steering direction with the VFH object based on the input laser scan data. Convert the steering direction to a desired linear and an angular velocity. If a steering direction is not found, the robot stops and searches by rotating in place.

Drive the robot by sending a message containing the angular velocity and the desired linear velocity using the ROS publisher.

```
while rate.TotalElapsedTime < 30
```

```
    % Get laser scan data
```

```
laserScan = receive(laserSub);
ranges = double(laserScan.Ranges);
angles = double(laserScan.readScanAngles);

% Create a lidarScan object from the ranges and angles
    scan = lidarScan(ranges,angles);

% Call VFH object to computer steering direction
steerDir = vfh(scan, targetDir);

% Calculate velocities
if ~isnan(steerDir) % If steering direction is valid
    desiredV = 0.2;
    w = exampleHelperComputeAngularVelocity(steerDir, 1);
else % Stop and search for valid direction
    desiredV = 0.0;
    w = 0.5;
end

% Assign and send velocity commands
velMsg.Linear.X = desiredV;
velMsg.Angular.Z = w;
velPub.send(velMsg);
end
```

This code shows how you can use the Navigation Toolbox™ algorithms to control robots and react to dynamic changes in their environment. Currently the loop ends after 30 seconds, but other conditions can be set to exit the loop based on information on the ROS network (i.e. robot position or number of laser scan messages).

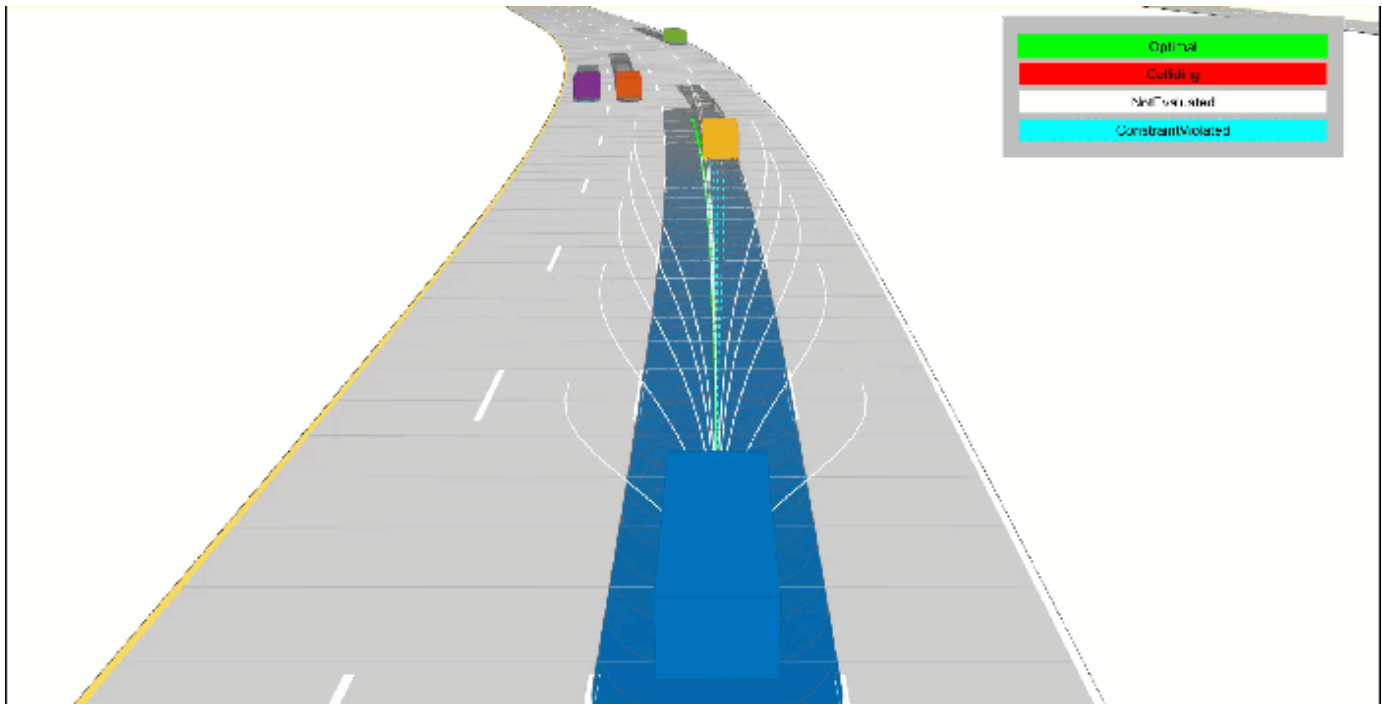
Disconnect from the ROS network

```
roshutdown
```

```
Shutting down global node /matlab_global_node_90736 with NodeURI http://192.168.233.1:61063/
```

Highway Trajectory Planning Using Frenet Reference Path

This example demonstrates how to plan a local trajectory in a highway driving scenario. This example uses a reference path and dynamic list of obstacles to generate alternative trajectories for an ego vehicle. The ego vehicle navigates through traffic defined in a provided driving scenario from a `drivingScenario` object. The vehicle alternates between adaptive cruise control, lane changing, and vehicle following maneuvers based on cost, feasibility, and collision-free motion.



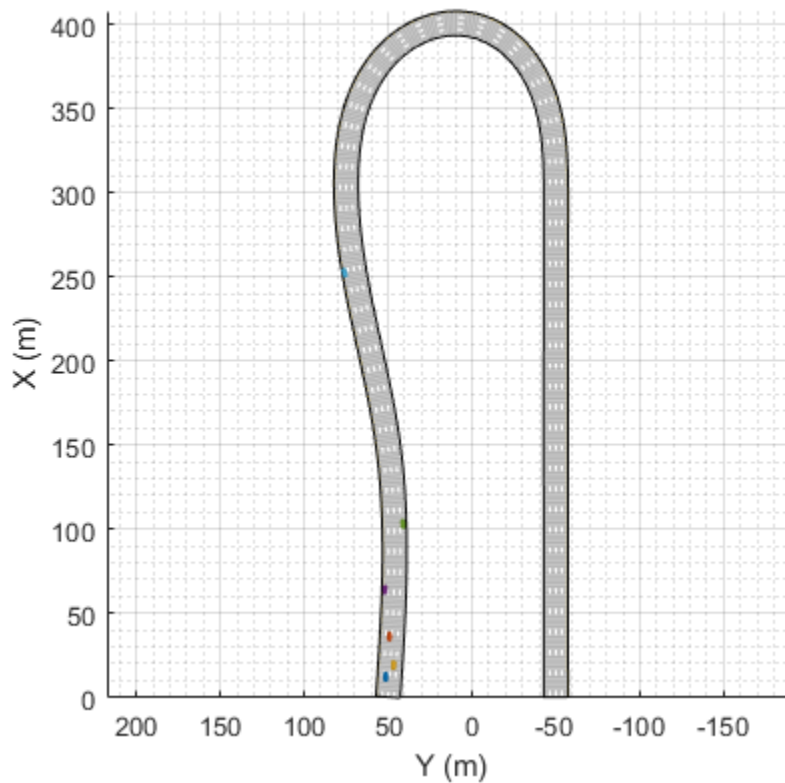
Load Driving Scenario

Begin by loading the provided `drivingScenario` object, which defines the vehicle and road properties in the current workspace. This scenario was generated using the Driving Scenario Designer (Automated Driving Toolbox) app and exported to a MATLAB® function, `drivingScenarioTrafficExample`. For more information, see “Create Driving Scenario Variations Programmatically” (Automated Driving Toolbox).

```
scenario = drivingScenarioTrafficExample;
% Default car properties
carLen   = 4.7; % in meters
carWidth = 1.8; % in meters
rearAxleRatio = .25;

% Define road dimensions
laneWidth  = carWidth*2; % in meters

plot(scenario);
```

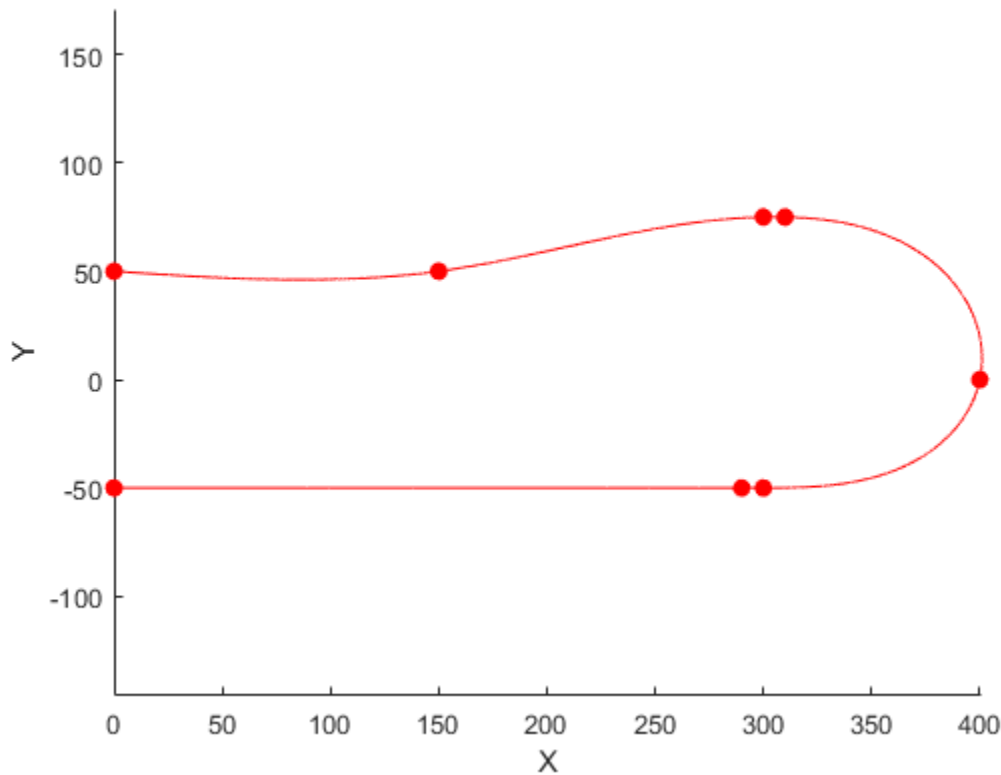


Construct Reference Path

All of the local planning in this example is performed with respect to a reference path, represented by a `referencePathFrenet` object. This object can return the state of the curve at given lengths along the path, find the closest point along the path to some global xy -location, and facilitates the coordinate transformations between global and Frenet reference frames.

For this example, the reference path is treated as the center of a four-lane highway, and the waypoints match the road defined in the provided `drivingScenario` object.

```
waypoints = [0 50; 150 50; 300 75; 310 75; 400 0; 300 -50; 290 -50; 0 -50]; % in meters
refPath = referencePathFrenet(waypoints);
ax = show(refPath);
axis(ax, 'equal'); xlabel('X'); ylabel('Y');
```



Construct Trajectory Generator

For a local planner, the goal is typically to sample a variety of possible motions that move towards a final objective while satisfying the current kinematic and dynamic conditions. The `trajectoryGeneratorFrenet` object accomplishes this by connecting the initial state with a set of terminal states using 4th- or 5th-order polynomial trajectories. Initial and terminal states are defined in the Frenet coordinate system, and each polynomial solution satisfies the lateral and longitudinal position, velocity, and acceleration boundary conditions while minimizing jerk.

Terminal states are often calculated using custom behaviors. These behaviors leverage information found in the local environment, such as the lane information, speed limit, and current or future predictions of actors in the ego vehicle's vicinity.

Construct a `trajectoryGeneratorFrenet` object using the reference path

```
connector = trajectoryGeneratorFrenet(refPath);
```

Construct Dynamic Collision Checker

The `dynamicCapsuleList` object is a data structure that represents the state of a dynamic environment over a discrete set of timesteps. This environment can then be used to efficiently validate multiple potential trajectories for the ego vehicle. Each object in the scene is represented by:

- Unique integer-valued identifier
- Properties for a capsule geometry used for efficient collision checking

- Sequence of SE2 states, where each state represents a discrete snapshot in time.

In this example, the trajectories generated by the `trajectoryGeneratorFrenet` object occur over some span of time, known as the time horizon. To ensure that collision checking covers all possible trajectories, the `dynamicCapsuleList` object should contain predicted trajectories of all actors spanning the maximum expected time horizon.

```
capList = dynamicCapsuleList;
```

Create a geometry structure for the ego vehicle with the given parameters.

```
egoID = 1;  
[egoID, egoGeom] = egoGeometry(capList, egoID);  
  
egoGeom.Geometry.Length = carLen; % in meters  
egoGeom.Geometry.Radius = carWidth/2; % in meters  
egoGeom.Geometry.FixedTransform(1, end) = -carLen*rearAxleRatio; % in meters
```

Add the ego vehicle to the dynamic capsule list.

```
updateEgoGeometry(capList, egoID, egoGeom);
```

Add the `drivingScenario` actors to the `dynamicCapsuleList` object. The geometry is set here, and the states are defined during the planning loop. You can see that the `dynamicCapsuleList` now contains one ego vehicle and five obstacles.

```
actorID = (1:5)';  
actorGeom = repelem(egoGeom, 5, 1);  
updateObstacleGeometry(capList, actorID, actorGeom)
```

```
ans =  
dynamicCapsuleList with properties:
```

```
    MaxNumSteps: 31  
      EgoIDs: 1  
  ObstacleIDs: [5x1 double]  
  NumObstacles: 5  
     NumEgos: 1
```

Planning Adaptive Routes Through Traffic

The planning utilities support a local planning strategy that samples a set of local trajectories based on the current and foreseen state of the environment before choosing the most optimal trajectory. The simulation loop has been organized into the following sections:

- 1 Advance the Ground Truth Scenario on page 1-0
- 2 Generate Terminal States on page 1-0
- 3 Evaluate Cost of Terminal States on page 1-0
- 4 Generate Trajectories and Check for Kinematic Feasibility on page 1-0
- 5 Check Trajectories for Collision and Select the Optimal Trajectory on page 1-0
- 6 Display the Sampled Trajectories and Animate Scene on page 1-0

Click the titles of each section to navigate to the relevant code in the simulation loop.

Advance Ground Truth Scenario on page 1-0

When planning in a dynamic environment, it is often necessary to estimate the state of the environment or predict its state in the near future. For simplicity, this example uses the `drivingScenario` as a ground truth source of trajectories for each actor over the planning horizon. To test a custom prediction algorithm, you can replace or modify the `exampleHelperRetrieveActorGroundTruth` function with custom code.

Generate Terminal States on page 1-0

A common goal in automated driving is to ensure that planned trajectories are not just feasible but also natural. Typical highway driving involves elements of lane keeping, maintaining the speed limit, changing lanes, adapting speed to traffic, and so on. Each custom behavior might require different environment information. This example demonstrates how to generate terminal states that implement three such behaviors: cruise control, lane changes, and follow lead vehicle.

Cruise control

The `exampleHelperBasicCruiseControl` function generates terminal states that carry out a cruise control behavior. This function uses the ego vehicle's lateral velocity and a time horizon to predict the ego vehicle's expected lane N -seconds in the future. The lane-center is calculated and becomes the terminal state's lateral deviation, and the lateral velocity and acceleration are set to zero.

For longitudinal boundary conditions, the terminal velocity is set to the road speed limit and the terminal acceleration is set to zero. The longitudinal position is unrestricted, which is specified as `NaN`. By dropping the longitude constraint, `trajectoryGeneratorFrenet` can use a lower 4th-order polynomial to solve the longitudinal boundary-value problem, resulting in a trajectory that smoothly matches the road speed over the given time horizon:

$$cruiseControlState = [\text{NaN } \dot{s}_{des} \ 0 \ l_{expLane} \ 0 \ 0]$$

Lane change

The `exampleHelperBasicLaneChange` function generates terminal states that transition the vehicle from the current lane to either adjacent lane. The function first determines the ego vehicle's current lane, and then checks whether the left and right lanes exist. For each existing lane, the terminal state is defined in the same manner as the cruise control behavior, with the exception that the terminal velocity is set to the current velocity rather than the road's speed limit:

$$laneChangeState = [\text{NaN } \dot{s}_{cur} \ 0 \ l_{desLane} \ 0 \ 0]$$

Follow lead vehicle

The `exampleHelperBasicLeadVehicleFollow` function generates terminal states that attempt to trail a vehicle found ahead of the ego vehicle. The function first determines the ego vehicle's current lane. For each provided `timeHorizon`, the function predicts the future state of each actor, finds all actors that occupy the same lane as the ego vehicle, and determines which is the closest *lead* vehicle (if no lead vehicles are found, the function does not return anything).

The ego vehicle's terminal state is calculated by taking the lead vehicle's position and velocity and reducing the terminal longitudinal position by some safety distance:

$$closestLeadVehicleState = [s_{lead} \ \dot{s}_{lead} \ 0 \ l_{lead} \ \dot{l}_{lead} \ 0]$$

$$followState = [(s_{lead} - d_{safety}) \dot{s}_{lead} \ 0 \ l_{lead} \ \dot{l}_{lead} \ 0]$$

Evaluate Cost of Terminal States on page 1-0

After the terminal states have been generated, their cost can be evaluated. Trajectory evaluation and the ways to prioritize potential solutions is highly subjective. For the sake of simplicity, the `exampleHelperEvaluateTSCost` function defines cost as the combination of three weighted sums.

- **Lateral Deviation Cost (C_{latDev})** — A positive weight that penalizes states that deviate from the center of a lane.

$$C_{latDev} = w_{\Delta L} * \Delta L$$

$$\Delta L = \operatorname{argmin}_i (|L_{termState} - L_{lane_i}|)$$

- **Time Cost (C_{time})** — A negative weight that prioritizes motions that occur over a longer interval, resulting in smoother trajectories.

$$C_{time} = w_{\Delta t} * \Delta t$$

- **Terminal Velocity Cost (C_{speed})** — A positive weight that prioritizes motions that maintain the speed limit, resulting in less dynamic maneuvers.

$$C_{speed} = w_{\Delta v} * \Delta v$$

Generate Trajectories and Check for Kinematic Feasibility on page 1-0

In addition to having terminal states with minimal cost, an optimal trajectory must often satisfy additional constraints related to kinematic feasibility and ride comfort. Trajectory constraints are one way of enforcing a desired ride quality, but they do so at the expense of a reduced driving envelope.

In this example, the `exampleHelperEvaluateTrajectory` function verifies that each trajectory satisfies the following constraints:

- **MaxAcceleration:** The absolute acceleration throughout the trajectory must be below a specified value. Smaller values reduce driving aggressiveness and eliminate kinematically infeasible trajectories. This restriction may eliminate maneuvers that could otherwise be performed by the vehicle.
- **MaxCurvature:** The minimum turning radius that is allowed throughout a trajectory. As with `MaxAcceleration`, reducing this value results in a smoother driving experience but may eliminate otherwise feasible trajectories.
- **MinVelocity:** This example constrains the ego vehicle to forward-only motion by setting a minimum velocity. This restriction is desired in highway driving scenarios and eliminates trajectories that fit overconstrained or poorly conditioned boundary values.

Check Trajectories for Collision and Select Optimal Trajectory on page 1-0

The final step in the planning process is choosing the best trajectory that also results in a collision-free path. Collision checking is often deferred until the end because it is an expensive operation, so by evaluating cost and analyzing constraints first, invalid trajectories can be removed from consideration. Remaining trajectories can then be checked for collision in optimal order until a collision free path has been found or all trajectories have been evaluated.

Define Simulator and Planning Parameters

This section defines the properties required to run the simulator and parameters that are used by the planner and behavior utilities. Properties such as `scenario.SampleTime` and `connector.TimeResolution` are synced so that states in the ground truth actor trajectories and planned ego trajectories occur at the same timesteps. Similarly, `replanRate`, `timeHorizons`, and `maxHorizon` are chosen such that they are integer multiples of the simulation rate.

As mentioned in the previous section, weights and constraints are selected to promote smooth driving trajectories while adhering to the rules of the road.

Lastly, define the `speedLimit` and `safetyGap` parameters, which are used to generate terminal states for the planner.

```
% Synchronize the simulator's update rate to match the trajectory generator's
% discretization interval.
scenario.SampleTime = connector.TimeResolution; % in seconds
```

```
% Define planning parameters.
replanRate = 10; % Hz
```

```
% Define the time intervals between current and planned states.
timeHorizons = 1:3; % in seconds
maxHorizon = max(timeHorizons); % in seconds
```

```
% Define cost parameters.
latDevWeight = 1;
timeWeight = -1;
speedWeight = 1;
```

```
% Reject trajectories that violate the following constraints.
maxAcceleration = 15; % in meters/second^2
maxCurvature = 1; % 1/meters, or radians/meter
minVelocity = 0; % in meters/second
```

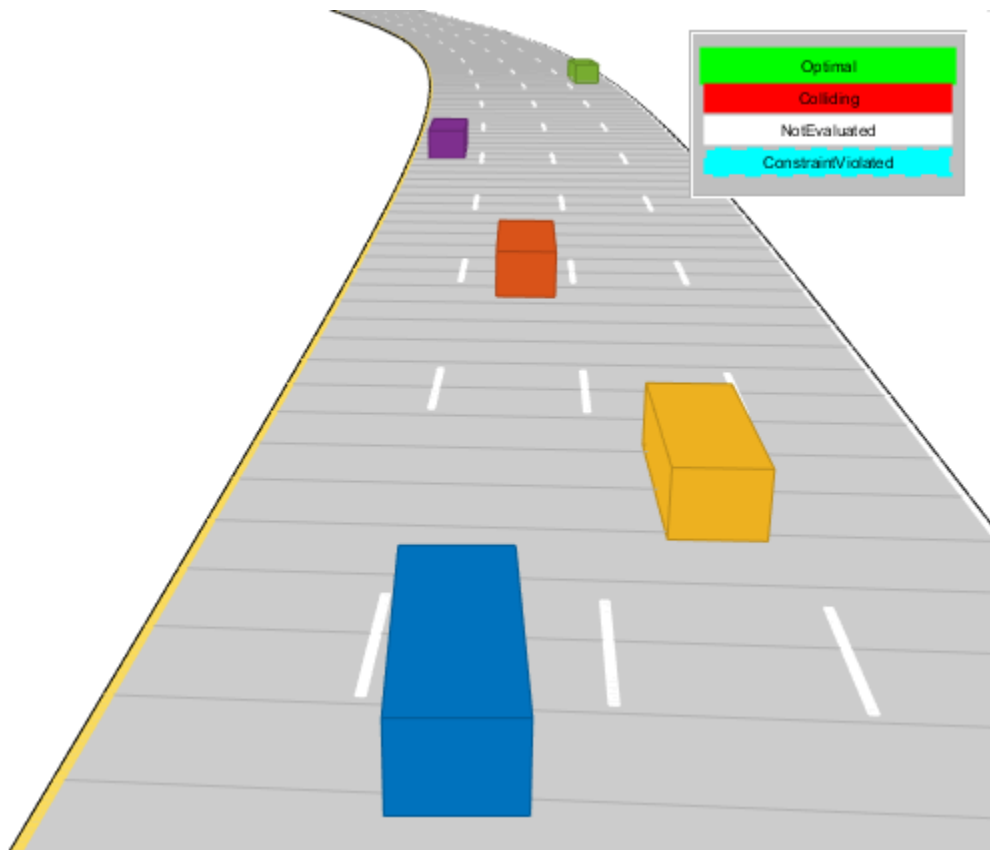
```
% Desired velocity setpoint, used by the cruise control behavior and when
% evaluating the cost of trajectories.
speedLimit = 11; % in meters/second
```

```
% Minimum distance the planner should target for following behavior.
safetyGap = 10; % in meters
```

Initialize Simulator

Initialize the simulator and create a `chasePlot` (Automated Driving Toolbox) viewer.

```
[scenarioViewer, futureTrajectory, actorID, actorPoses, egoID, egoPoses, stepPerUpdate, egoState, isRunn
    exampleHelperInitializeSimulator(scenario, capList, refPath, laneWidth, replanRate, carLen);
```



Run Driving Simulation

```

tic
while isRunning
    % Retrieve the current state of actor vehicles and their trajectory over
    % the planning horizon.
    [curActorState, futureTrajectory, isRunning] = ...
        exampleHelperRetrieveActorGroundTruth(scenario, futureTrajectory, replanRate, maxHorizon);

    % Generate cruise control states.
    [termStatesCC, timesCC] = exampleHelperBasicCruiseControl(...
        refPath, laneWidth, egoState, speedLimit, timeHorizons);

    % Generate lane change states.
    [termStatesLC, timesLC] = exampleHelperBasicLaneChange(...
        refPath, laneWidth, egoState, timeHorizons);

    % Generate vehicle following states.
    [termStatesF, timesF] = exampleHelperBasicLeadVehicleFollow(...
        refPath, laneWidth, safetyGap, egoState, curActorState, timeHorizons);

    % Combine the terminal states and times.
    allTS = [termStatesCC; termStatesLC; termStatesF];
    allDT = [timesCC; timesLC; timesF];
    numTS = [numel(timesCC); numel(timesLC); numel(timesF)];

    % Evaluate cost of all terminal states.

```

```

costTS = exampleHelperEvaluateTSCost(allTS,allDT,laneWidth,speedLimit,...
    speedWeight, latDevWeight, timeWeight);

% Generate trajectories.
egoFrenetState = global2frenet(refPath,egoState);
[frenetTraj,globalTraj] = connect(connector,egoFrenetState,allTS,allDT);

% Eliminate trajectories that violate constraints.
isValid = exampleHelperEvaluateTrajectory(globalTraj,maxAcceleration,maxCurvature,minVelocity);

% Update the collision checker with the predicted trajectories
% of all actors in the scene.
for i = 1:numel(actorPoses)
    actorPoses(i).States = futureTrajectory(i).Trajectory(:,1:3);
end
updateObstaclePose(capList,actorID,actorPoses);

% Determine evaluation order.
[cost, idx] = sort(costTS);
optimalTrajectory = [];

trajectoryEvaluation = nan(numel(isValid),1);

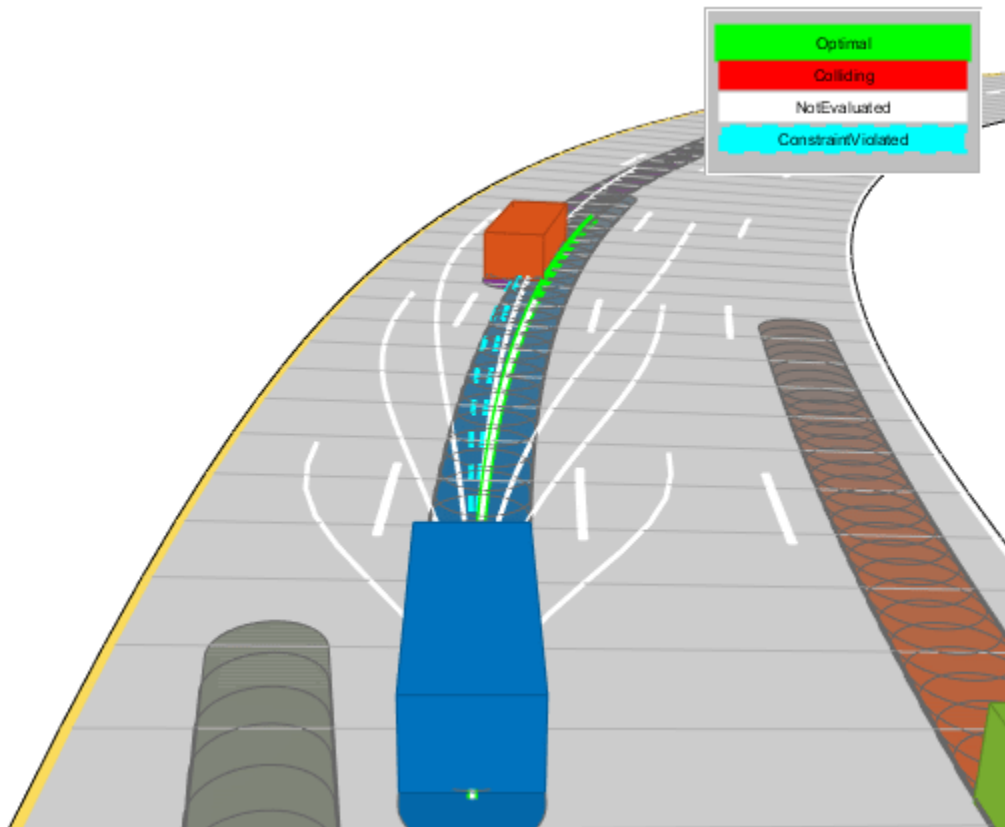
% Check each trajectory for collisions starting with least cost.
for i = 1:numel(idx)
    if isValid(idx(i))
        % Update capsule list with the ego object's candidate trajectory.
        egoPoses.States = globalTraj(idx(i)).Trajectory(:,1:3);
        updateEgoPose(capList,egoID,egoPoses);

        % Check for collisions.
        isColliding = checkCollision(capList);

        if all(~isColliding)
            % If no collisions are found, this is the optimal.
            % trajectory.
            trajectoryEvaluation(idx(i)) = 1;
            optimalTrajectory = globalTraj(idx(i)).Trajectory;
            break;
        else
            trajectoryEvaluation(idx(i)) = 0;
        end
    end
end

% Display the sampled trajectories.
lineHandles = exampleHelperVisualizeScene(lineHandles,globalTraj,isValid,trajectoryEvaluation);

```



```

hold on;
show(capList,'TimeStep',1:capList.MaxNumSteps,'FastUpdate',1);
hold off;

if isempty(optimalTrajectory)
    % All trajectories either violated a constraint or resulted in collision.
    %
    % If planning failed immediately, revisit the simulator, planner,
    % and behavior properties.
    %
    % If the planner failed midway through a simulation, additional
    % behaviors can be introduced to handle more complicated planning conditions.
    error('No valid trajectory has been found.');
```

```

else
    % Visualize the scene between replanning.
    for i = (2+(0:(stepPerUpdate-1)))
        % Approximate realtime visualization.
        dt = toc;
        if scenario.SampleTime-dt > 0
            pause(scenario.SampleTime-dt);
        end

        egoState = optimalTrajectory(i,:);
        scenarioViewer.Actors(1).Position(1:2) = egoState(1:2);
        scenarioViewer.Actors(1).Velocity(1:2) = [cos(egoState(3)) sin(egoState(3))]*egoState(3);
        scenarioViewer.Actors(1).Yaw = egoState(3)*180/pi;
    end
end

```

```

scenarioViewer.actors(1).AngularVelocity(3) = egoState(4)*egoState(5);

% Update capsule visualization.
hold on;
show(capList,'TimeStep',i:capList.MaxNumSteps,'FastUpdate',1);
hold off;

% Update driving scenario.
advance(scenarioViewer);
tic;
end
end
end

```

Planner Customizations and Additional Considerations

Custom solutions often involve many tunable parameters, each capable of changing the final behavior in ways that are difficult to predict. This section highlights some of the feature-specific properties and their effect on the above planner. Then, suggestions provide ways to tune or augment the custom logic.

Dynamic Capsule List

As mentioned previously, the `dynamicCapsuleList` object acts as a temporal database, which caches predicted trajectories of obstacles. You can perform collision checking with one or more ego bodies over some span of time. The `MaxNumSteps` property determines the total number of time-steps that are checked by the object. In the above simulation loop, the property was set to 31. This value means the planner checks the entire 1-3 second span of any trajectories (sampled at every 0.1 second). Now, increase the maximum value in `timeHorizons`:

```

timeHorizons = 1:5; % in seconds
maxTimeHorizon = max(timeHorizons); % in seconds

```

There are now two options:

- 1 The `MaxNumSteps` property is left unchanged.
- 2 The `MaxNumSteps` property is updated to accommodate the new max timespan.

If the property is left unchanged, then the capsule list only validates the first 3 seconds of any trajectory, which may be preferable if computational efficiency is paramount or the prediction certainty drops off quickly.

Alternatively, one may be working with ground truth data (as is shown above), or the future state of the environment is well known (e.g. a fully automated environment with centralized control). Since this example uses ground truth data for the actors, update the property.

```

capList.MaxNumSteps = 1+floor(maxTimeHorizon/scenario.SampleTime);

```

Another, indirectly tunable, property of the list is the capsule geometry. The geometry of the ego vehicle or actors can be inflated by increasing the `Radius`, and buffer regions can be added to vehicles by modifying the `Length` and `FixedTransform` properties.

Inflate the ego vehicle's entire footprint by increasing the radius.

```

egoGeom.Geometry.Radius = laneWidth/2; % in meters
updateEgoGeometry(capList,egoID,egoGeom);

```

Add a front and rear buffer region to all actors.

```
actorGeom(1).Geometry.Length = carLen*1.5; % in meters
actorGeom(1).Geometry.FixedTransform(1,end) = -actorGeom(1).Geometry.Length*rearAxleRatio; % in m
actorGeom = repmat(actorGeom(1),5,1);
updateObstacleGeometry(capList,actorID,actorGeom);
```

Rerun Simulation With Updated Properties

Rerun the simulation. The resulting simulation has a few interesting developments:

- The longer five-second time horizon results in a much smoother driving experience. The planner still prioritizes the longer trajectories due to the negative `timeWeight`.
- The updated `MaxNumSteps` property has enabled collision checking over the full trajectory. When paired with the longer planning horizon, the planner identifies and discards the previously optimal left-lane change and returns to the original lane.
- The inflated capsules find a collision earlier and reject a trajectory, which results in more conservative driving behavior. One potential drawback to this is a reduced planning envelope, which runs the risk of the planner not being able to find a valid trajectory.

```
% Initialize the simulator and create a chasePlot viewer.
[scenarioViewer, futureTrajectory, actorID, actorPoses, egoID, egoPoses, stepPerUpdate, egoState, isRunning] =
    exampleHelperInitializeSimulator(scenario, capList, refPath, laneWidth, replanRate, carLen);
tic;
while isRunning
    % Retrieve the current state of actor vehicles and their trajectory over
    % the planning horizon.
    [curActorState, futureTrajectory, isRunning] = exampleHelperRetrieveActorGroundTruth(...
        scenario, futureTrajectory, replanRate, maxHorizon);

    % Generate cruise control states.
    [termStatesCC, timesCC] = exampleHelperBasicCruiseControl(...
        refPath, laneWidth, egoState, speedLimit, timeHorizons);

    % Generate lane change states.
    [termStatesLC, timesLC] = exampleHelperBasicLaneChange(...
        refPath, laneWidth, egoState, timeHorizons);

    % Generate vehicle following states.
    [termStatesF, timesF] = exampleHelperBasicLeadVehicleFollow(...
        refPath, laneWidth, safetyGap, egoState, curActorState, timeHorizons);

    % Combine the terminal states and times.
    allTS = [termStatesCC; termStatesLC; termStatesF];
    allDT = [timesCC; timesLC; timesF];
    numTS = [numel(timesCC); numel(timesLC); numel(timesF)];

    % Evaluate cost of all terminal states.
    costTS = exampleHelperEvaluateTSCost(allTS, allDT, laneWidth, speedLimit, ...
        speedWeight, latDevWeight, timeWeight);

    % Generate trajectories.
    egoFrenetState = global2frenet(refPath, egoState);
    [frenetTraj, globalTraj] = connect(connector, egoFrenetState, allTS, allDT);

    % Eliminate trajectories that violate constraints.
    isValid = exampleHelperEvaluateTrajectory(...
```



```

    globalTraj, maxAcceleration, maxCurvature, minVelocity);

% Update the collision checker with the predicted trajectories
% of all actors in the scene.
for i = 1:numel(actorPoses)
    actorPoses(i).States = futureTrajectory(i).Trajectory(:,1:3);
end
updateObstaclePose(capList, actorID, actorPoses);

% Determine evaluation order.
[cost, idx] = sort(costTS);
optimalTrajectory = [];

trajectoryEvaluation = nan(numel(isValid),1);

% Check each trajectory for collisions starting with least cost.
for i = 1:numel(idx)
    if isValid(idx(i))
        % Update capsule list with the ego object's candidate trajectory.
        egoPoses.States = globalTraj(idx(i)).Trajectory(:,1:3);
        updateEgoPose(capList, egoID, egoPoses);

        % Check for collisions.
        isColliding = checkCollision(capList);

        if all(~isColliding)
            % If no collisions are found, this is the optimal
            % trajectory.
            trajectoryEvaluation(idx(i)) = 1;
            optimalTrajectory = globalTraj(idx(i)).Trajectory;
            break;
        else
            trajectoryEvaluation(idx(i)) = 0;
        end
    end
end

% Display the sampled trajectories.
lineHandles = exampleHelperVisualizeScene(lineHandles, globalTraj, isValid, trajectoryEvaluation);

if isempty(optimalTrajectory)
    % All trajectories either violated a constraint or resulted in collision.
    %
    % If planning failed immediately, revisit the simulator, planner,
    % and behavior properties.
    %
    % If the planner failed midway through a simulation, additional
    % behaviors can be introduced to handle more complicated planning conditions.
    error('No valid trajectory has been found.');
```

```

else
    % Visualize the scene between replanning.
    for i = (2+(0:(stepPerUpdate-1)))
        % Approximate realtime visualization.
        dt = toc;
        if scenario.SampleTime-dt > 0
            pause(scenario.SampleTime-dt);
        end
    end
end

```

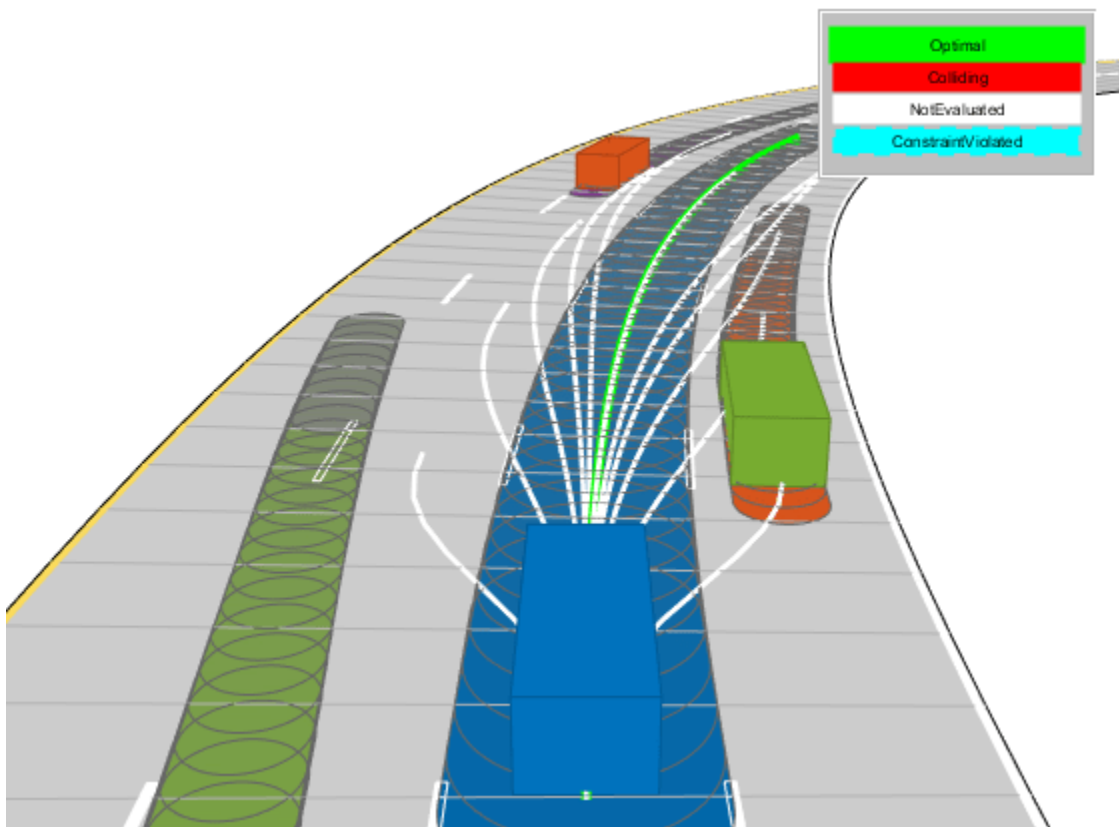
```

egoState = optimalTrajectory(i,:);
scenarioViewer.Actors(1).Position(1:2) = egoState(1:2);
scenarioViewer.Actors(1).Velocity(1:2) = [cos(egoState(3)) sin(egoState(3))]*egoState(4);
scenarioViewer.Actors(1).Yaw = egoState(3)*180/pi;
scenarioViewer.Actors(1).AngularVelocity(3) = egoState(4)*egoState(5);

% Update capsule visualization.
hold on;
show(capList, 'TimeStep', i:capList.MaxNumSteps, 'FastUpdate', 1);
hold off;

% Update driving scenario.
advance(scenarioViewer);
tic;
end
end
end
end

```



Optimal Trajectory Generation for Urban Driving

This example shows how to perform dynamic replanning in an urban scenario using `trajectoryOptimalFrenet`.

In this example, you will:

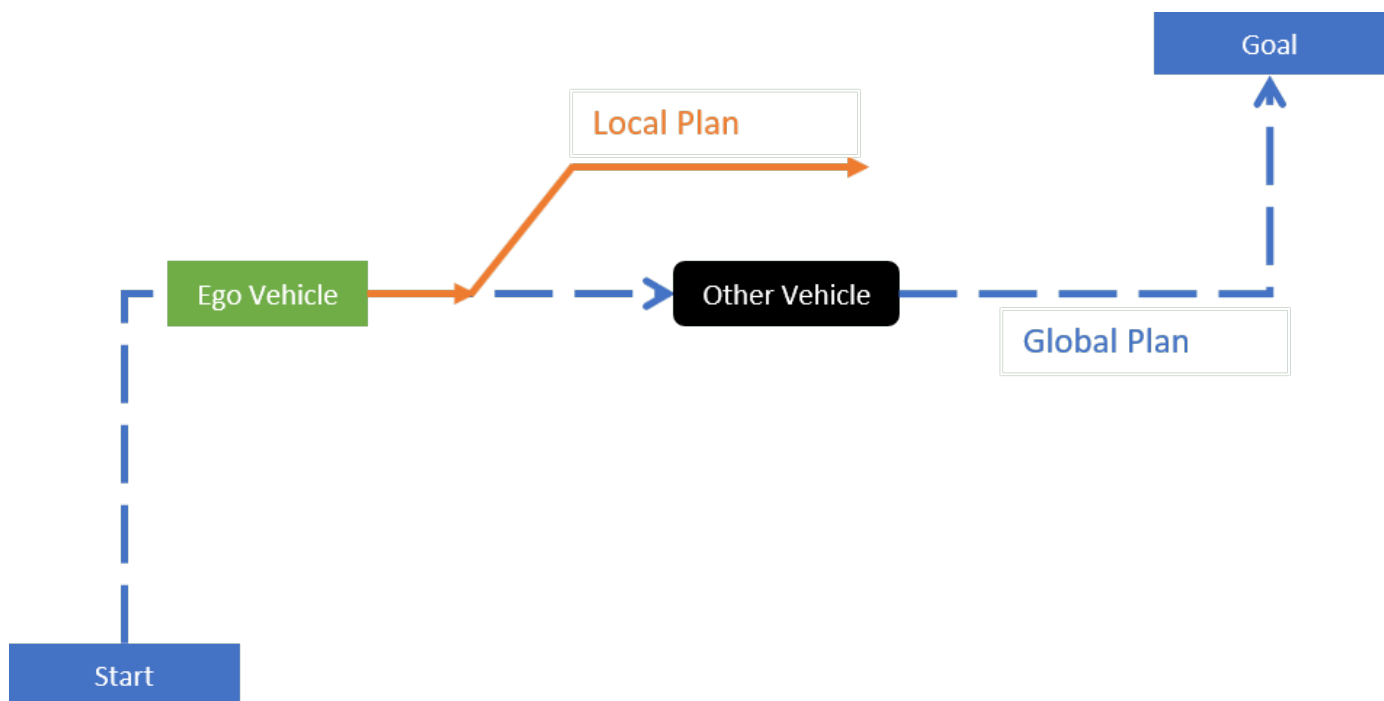
- 1 Explore an urban scenario with predefined vehicles.
- 2 Use `trajectoryOptimalFrenet` to do local planning for navigating the Urban scenario.

Contents

- Introduction on page 1-0
- Explore an Urban Scenario for Local planning on page 1-0
- Use `trajectoryOptimalFrenet` to demonstrate Adaptive Cruise Control (ACC) behavior on page 1-0
- Use `trajectoryOptimalFrenet` to negotiate a smooth turn on page 1-0
- Use `trajectoryOptimalFrenet` to perform Lane Change maneuver on page 1-0
- Conclusion on page 1-0

Introduction

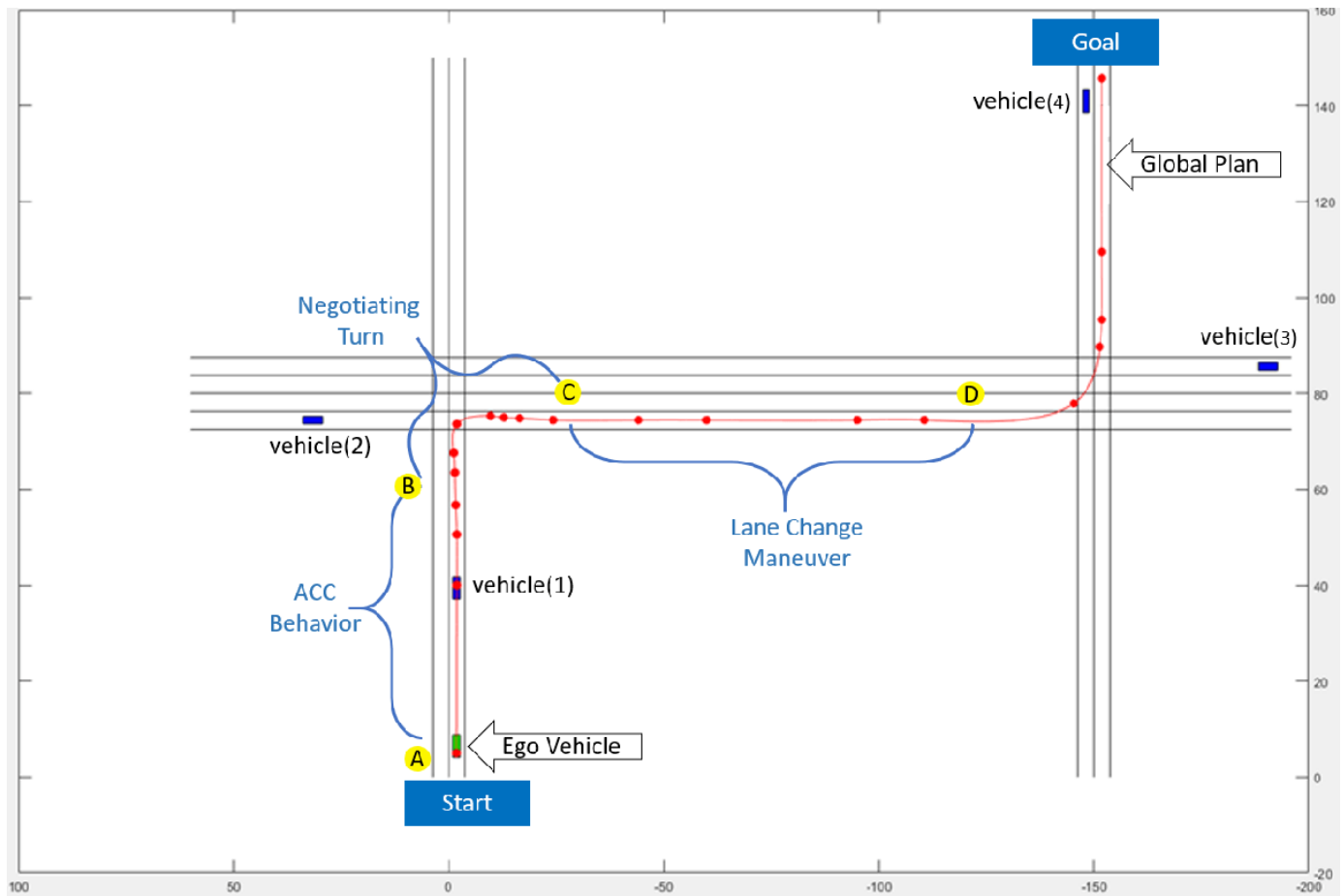
Automated driving in an urban scenario needs planning on two levels, global and local. The *global planner* finds the most feasible path between start and goal points. The *local planner* does dynamic replanning to generate an optimal trajectory based on the global plan and the surrounding information. In this example, an ego vehicle (green box) follows a global plan (dotted blue line). Local planning is done (solid orange line) while trying to avoid another vehicle (black rectangle).



Local planners use obstacle information to generate optimal collision-free trajectories. Obstacle information from various on-board sensors like camera, radar, and lidar are fused to keep the *occupancy map* updated. This occupancy map is *egocentric*, where the local frame is centered on the ego vehicle. The map is used for local planning when obstacles are detected from the sensors and placed on the map.

Explore An Urban Scenario For Local Planning

This example scenario has four other vehicles (blue rectangles), which are moving in predefined paths at different velocities. The figure below illustrates this scenario and the global plan (solid red line) used in this example. Solid red dots in the below figure represent the waypoints of the global plan between the start and goal positions. The green rectangle represents the ego vehicle.



The ego vehicle uses `trajectoryOptimalFrenet` to navigate from position **A** to position **D** in three segments with three different configuration parameters.

- First (**A** to **B**), the vehicle demonstrates Adaptive Cruise Control (ACC) behavior.
- Second (**B** to **C**), the vehicle negotiates a turn to follow the global plan.
- Third (**C** to **D**), the vehicle performs a lane change maneuver.

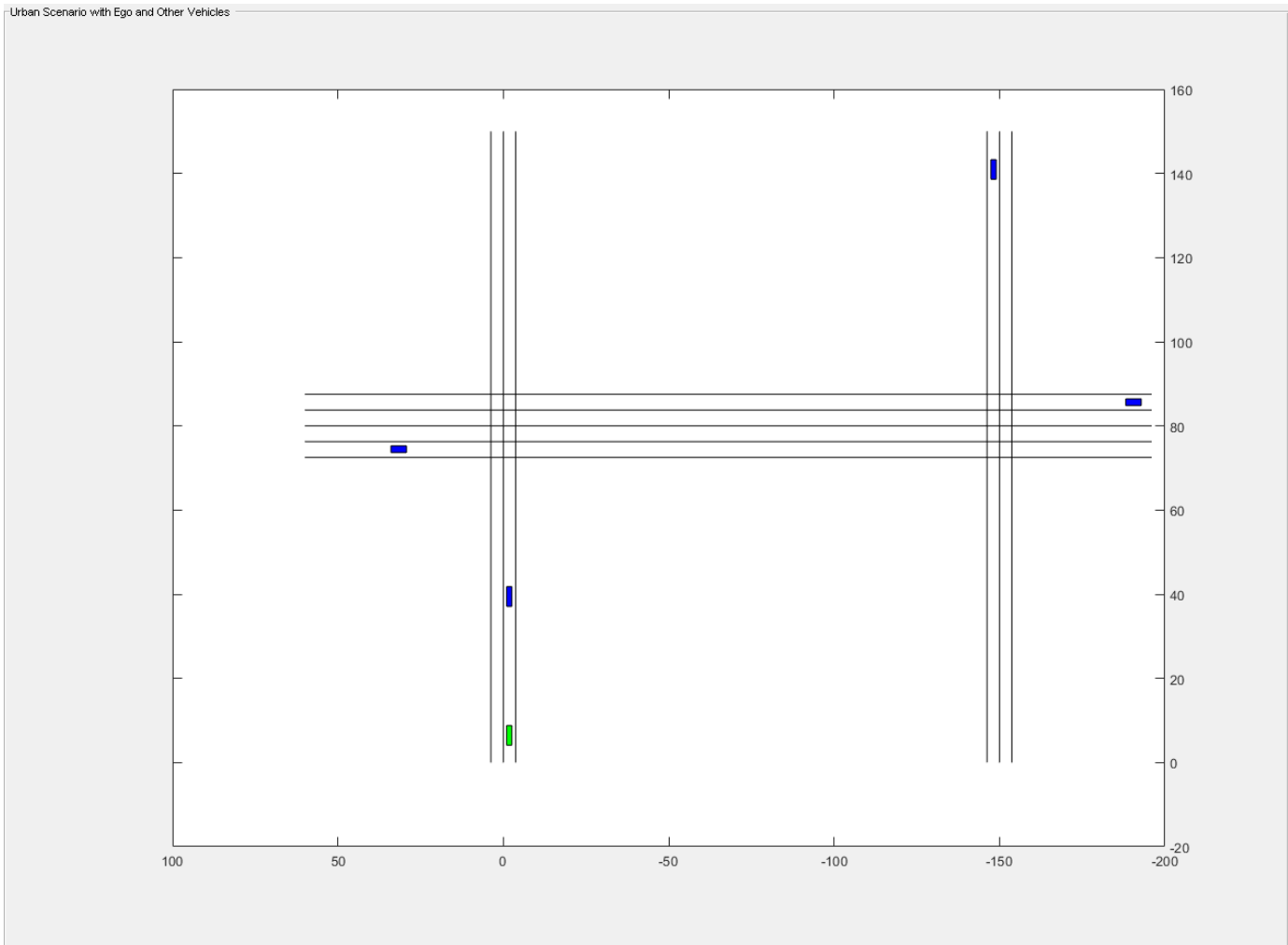
Set up the required data and environment variables:

```
% Load data from urbanScenarioData.mat file, initialize required variables
[otherVehicles,globalPlanPoints,stateValidator] = exampleHelperUrbanSetup;
```

- 1 otherVehicles:** [1 x 4] Structure array containing fields: Position, Yaw, Velocity, and SimulationTime, of each vehicle in the scenario.
- 2 globalPlanPoints:** [18 x 2] Matrix contains precomputed global plan consisting of eighteen waypoints, each representing a position in the scenario.
- 3 stateValidator:** validatorOccupancyMap object that acts as the state validator based on a given 2-D grip map. A fully occupied egocentric occupancy map is updated based on obstacle information and road boundaries. A custom state validator can also be used based on the application. For more information, see `nav.StateValidator`.

Plot the scenario.

```
exampleHelperPlotUrbanScenario;
```



Create local planner

Specify the state validator and global plan to create a local planner using `trajectoryOptimalFrenet`.

```
localPlanner = trajectoryOptimalFrenet(globalPlanPoints, stateValidator);
```

Explore properties of LocalPlanner

The `localPlanner` has a variety of properties that can be tuned to achieve the desired behavior. This section explores some of these properties and their default values.

`LocalPlanner.TerminalStates`

- **Longitudinal: [30 45 60 75 90]**: Defines longitudinal sampling distance in meters. This value can be a scalar or vector.
- **Lateral: [-2 -1 0 1 2]**: Defines lateral deviation in meters from the reference path (Global plan in our case).
- **Time: 7**: Time in seconds to reach the end of trajectory.
- **Speed: 10**: Velocity in meters per second, to be achieved at the end of the trajectory
- **Acceleration: 0**: Acceleration at the end of the trajectory in m/s^2 .

`LocalPlanner.FeasibilityParameters`

- **MaxCurvature: 0.1** : Maximum feasible value for the curvature in m^{-1}
- **MaxAcceleration: 2.5**: Maximum feasible acceleration in m/s^2 .

`LocalPlanner.TimeResolution: 0.1`: Trajectory discretization interval in seconds

Use `trajectoryOptimalFrenet` to demonstrate Adaptive Cruise Control (ACC) behavior

In this section, assign the properties needed to configure `localPlanner` to demonstrate Adaptive Cruise Control (ACC) behavior.

To demonstrate ACC, the ego vehicle needs to follow a lead vehicle by maintaining a safe distance. The lead vehicle in this segment is fetched using `otherVehicles(1)`.

```
% Get leadVehicle in segment from Position A to Position B
leadVehicle = otherVehicles(1);

% Define ACC safe distance
ACCsafeDistance = 35; % in meters
% Adjusting the time resolution of planner object to make the ego vehicle
% travel smoothly
timeResolution = 0.01;
localPlanner.TimeResolution = timeResolution;
```

Set up the ego vehicle at position A and define its initial velocity and orientation (Yaw).

```
% Set positions A, B, C and D
positionA = [5.1, -1.8, 0];
positionB = [60, -1.8, 0];
positionC = [74.45, -30.0, 0];
positionD = [74.45, -135, 0];
goalPoint = [145.70, -151.8, 0];

% Set the initial state of the ego vehicle
egoInitPose = positionA;
egoInitVelocity = [10, -0.3, 0];
egoInitYaw = -0.165;
```

```

currentEgoState = [egoInitPose(1), egoInitPose(2), deg2rad(egoInitYaw), ...
    0, norm(egoInitVelocity), 0];
vehicleLength = 4.7; % in meters
% Replan interval in number of simulation steps
% (default 50 simulation steps)
replanInterval = 50;

```

Visualize the simulation results.

```

% Initialize Visualization
exampleHelperInitializeVisualization;

```

The ACC behavior is achieved by setting the TerminalStates of localPlanner as below:

To maintain the safe distance from lead vehicle, set
`localPlanner.TerminalStates.Longitudinal` to Distance to Lead Vehicle - Vehicle Length;

To maintain relative velocity with respect to the lead vehicle, set
`localPlanner.TerminalStates.Speed` to Lead Vehicle Velocity;

To continue navigating on the global plan, set `localPlanner.TerminalStates.Lateral` to 0;

In the following code snippet, `localPlanner` generates trajectory that is executed and visualized using `exampleHelperUpdateVisualization` for every simulation step. However, replanning is done at every `replanInterval` simulation step. The following is the sequence of events during replanning:

- Update the occupancy map using vehicle information using `exampleHelperUpdateOccupancyMap`.
- Update the `localPlanner.TerminalStates`.
- Trajectory generation using `plan(localPlanner, currentStateInFrenet)`.

```

% Simulate till the ego vehicle reaches position B
simStep = 1;
% Check only for X as there is no change in Y.
while currentEgoState(1) <= positionB(1)

    % Replan at every "replanInterval"th simulation step
    if rem(simStep, replanInterval) == 0 || simStep == 1
        % Compute the replanning time
        previousReplanTime = simStep*timeResolution;

        % Updating occupancy map with vehicle information
        exampleHelperUpdateOccupancyMap(otherVehicles, simStep, currentEgoState);

        % Compute distance to Lead Vehicle and leadVehicleVelocity
        distanceToLeadVehicle = pdist2(leadVehicle.Position(simStep,1:2), ...
            currentEgoState(1:2));
        leadVehicleVelocity = leadVehicle.Velocity(simStep,:);

        % Set localPlanner.TerminalStates for ACC behavior
        if distanceToLeadVehicle <= ACCSafeDistance
            localPlanner.TerminalStates.Longitudinal = distanceToLeadVehicle - vehicleLength;
            localPlanner.TerminalStates.Speed = norm(leadVehicleVelocity);
            localPlanner.TerminalStates.Lateral = 0;
            desiredTimeBound = localPlanner.TerminalStates.Longitudinal/...
                localPlanner.TerminalStates.Speed;
        end
    end
end

```

```
        localPlanner.TerminalStates.Time = desiredTimeBound;
        localPlanner.FeasibilityParameters.MaxCurvature = 0.5;
    end

    % Generate optimal trajectory for current set of parameters
    currentStateInFrenet = cart2frenet(localPlanner, [currentEgoState(1:5) 0]);
    trajectory = plan(localPlanner, currentStateInFrenet);

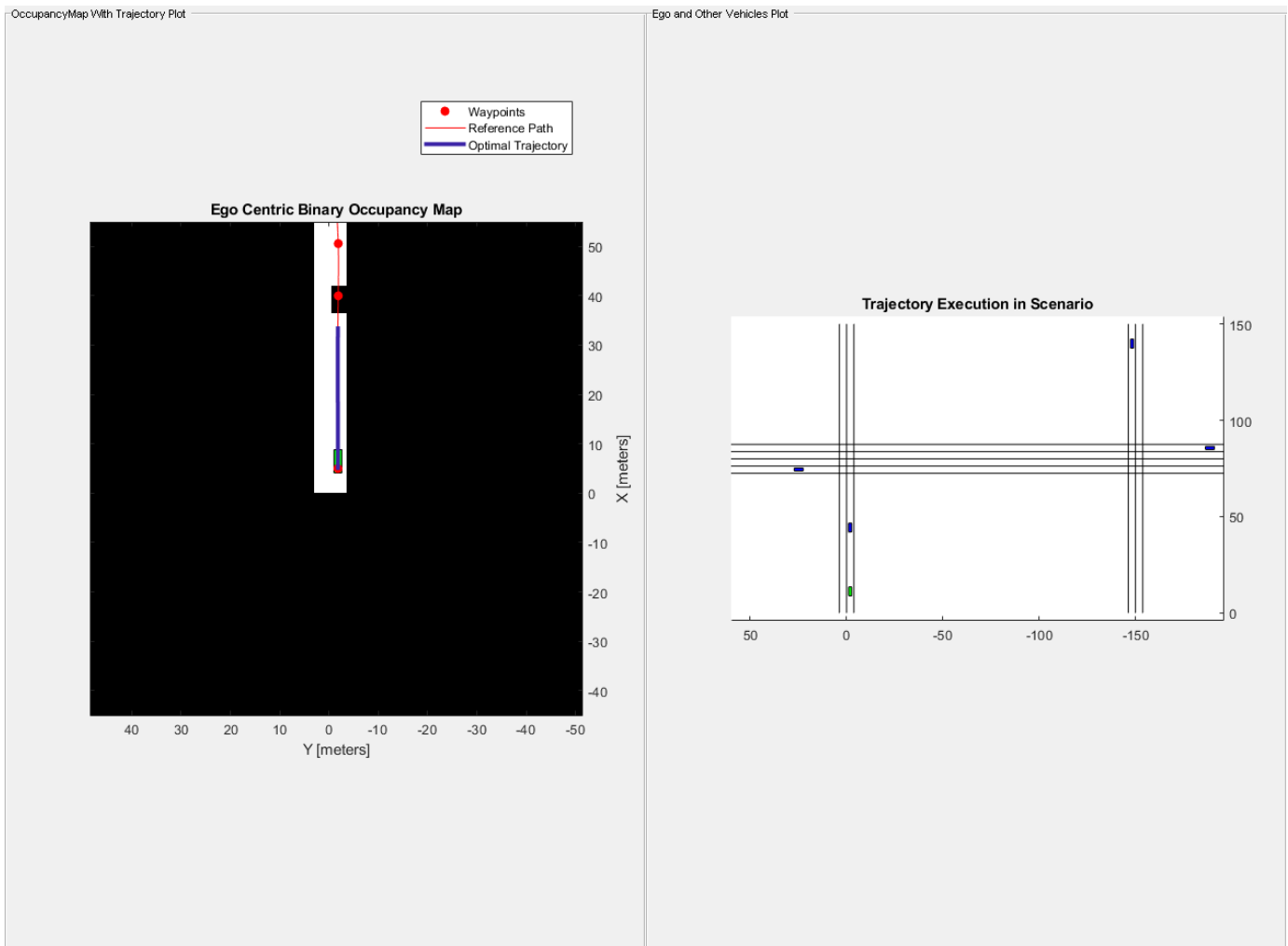
    % Visualize the ego-centric occupancy map
    show(egoMap, "Parent", hAxes1);
    hAxes1.Title.String = "Ego Centric Binary Occupancy Map";

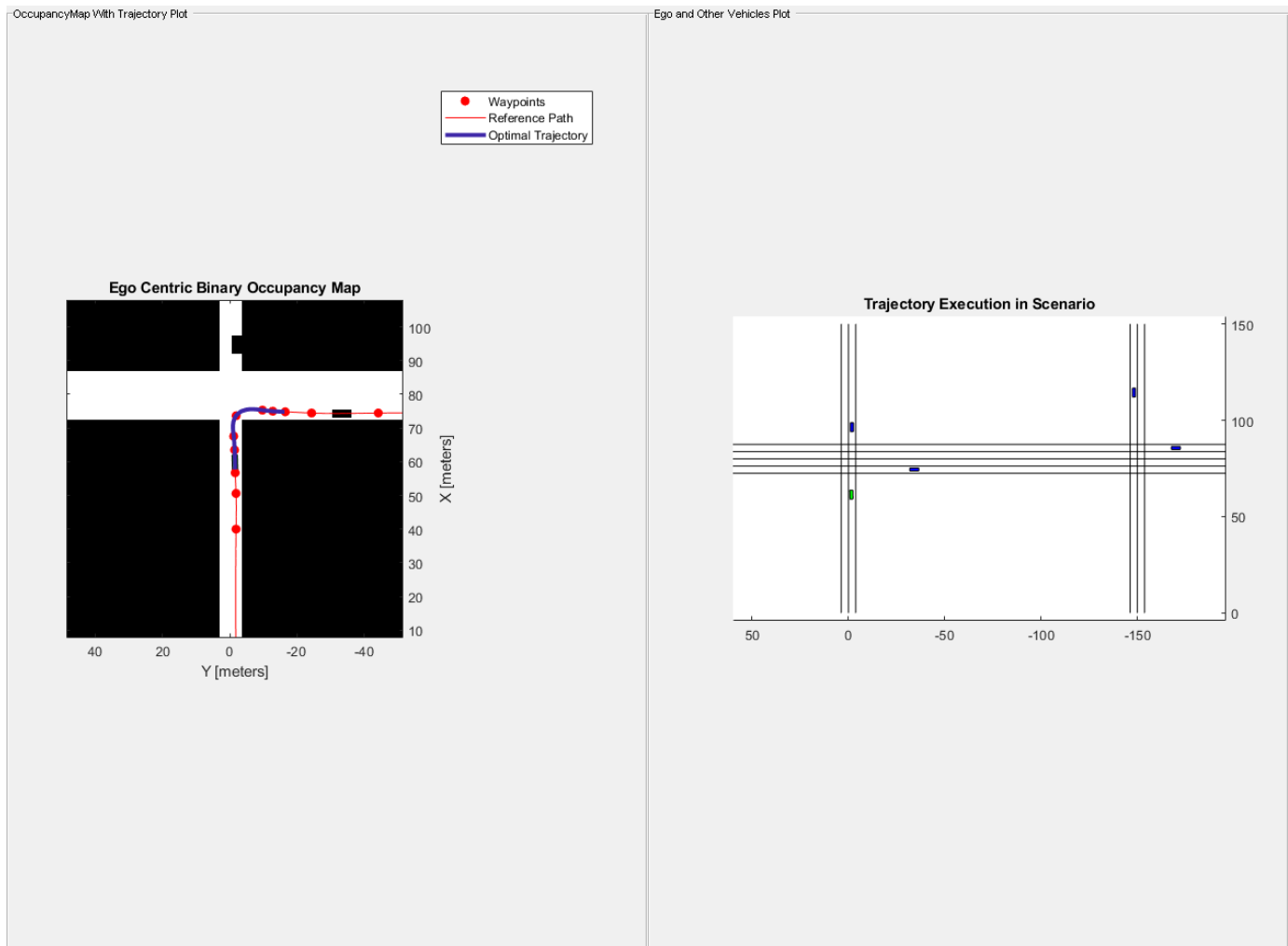
    % Visualize ego vehicle on occupancy map
    egoCenter = currentEgoState(1:2);
    egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)), egoCenter);
    hold(hAxes1, "on")
    fill(egoPolygon(1, :), egoPolygon(2, :), "g", "Parent", hAxes1)

    % Visualize the Trajectory reference path and trajectory
    show(localPlanner, "Trajectory", "optimal", "Parent", hAxes1)
end

% Execute and Update Visualization
[isGoalReached, currentEgoState] = ...
    exampleHelperExecuteAndVisualize(currentEgoState, simStep, ...
    trajectory, previousReplanTime);
if(isGoalReached)
    break;
end

% Update the simulation step for the next iteration
simStep = simStep + 1;
pause(0.01);
end
```



At the end of this execution, the ego vehicle is at position B.

Next, configure `trajectoryOptimalFrenet` for negotiating a turn in the second segment from position B to position C.

Use `trajectoryOptimalFrenet` to Negotiate a Smooth Turn

The current set properties of the `localPlanner` are sufficient to negotiate a smooth turn. However, in the second segment, the lead vehicle is fetched from `otherVehicles(2)`.

```
% Set Lead Vehicle to correspond to the vehicle in second segment
% from position B to position C
leadVehicle = otherVehicles(2);

% Simulate till the ego vehicle reaches position C
% Check only for Y as there is no change in X at C
while currentEgoState(2) >= positionC(2)

    % Replan at every "replanInterval"th simulation step
    if rem(simStep, replanInterval) == 0
        % Compute the replanning time
```

```

previousReplanTime = simStep*timeResolution;

% Updating occupancy map with vehicle information
exampleHelperUpdateOccupancyMap(otherVehicles, simStep, currentEgoState);

% Compute distance to Lead Vehicle and leadVehicleVelocity
distanceToLeadVehicle = pdist2(leadVehicle.Position(simStep,1:2), ...
    currentEgoState(1:2));
leadVehicleVelocity = leadVehicle.Velocity(simStep,:);

if(distanceToLeadVehicle <= ACCSafeDistance)
    localPlanner.TerminalStates.Longitudinal = distanceToLeadVehicle - vehicleLength;
    localPlanner.TerminalStates.Speed = norm(leadVehicleVelocity);
    localPlanner.TerminalStates.Lateral = 0;
    desiredTimeBound = localPlanner.TerminalStates.Longitudinal/...
        localPlanner.TerminalStates.Speed;
    localPlanner.TerminalStates.Time = desiredTimeBound;
    localPlanner.FeasibilityParameters.MaxCurvature = 0.5;
    localPlanner.FeasibilityParameters.MaxAcceleration = 5;
end

% Generate optimal trajectory for current set of parameters
currentStateInFrenet = cart2frenet(localPlanner, [currentEgoState(1:5) 0]);
trajectory = plan(localPlanner, currentStateInFrenet);

% Visualize the ego-centric occupancy map
show(egoMap, "Parent", hAxes1);
hAxes1.Title.String = "Ego Centric Binary Occupancy Map";

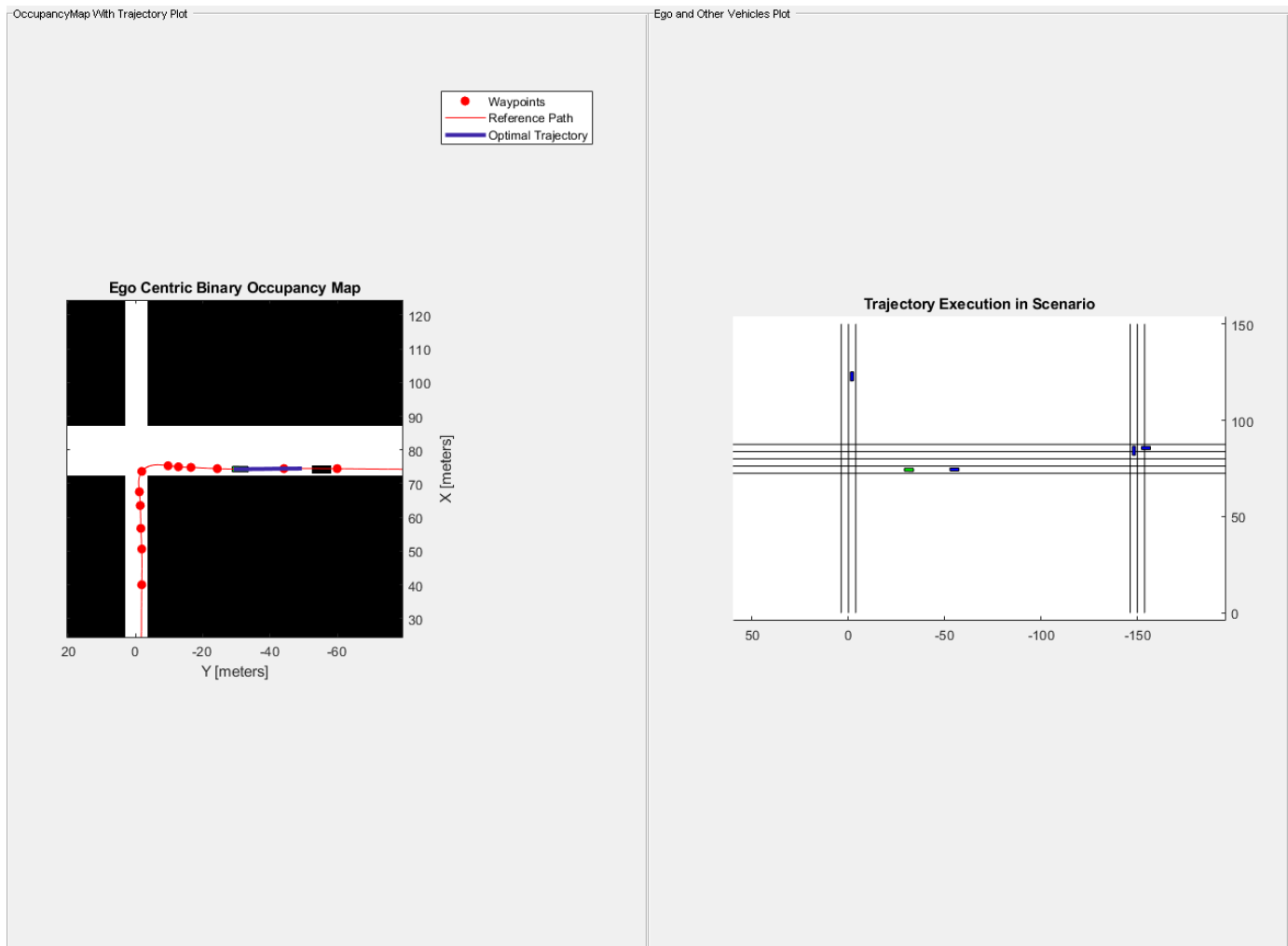
% Visualize ego vehicle on occupancy map
egoCenter = currentEgoState(1:2);
egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)), egoCenter,
    hold(hAxes1, "on")
    fill(egoPolygon(1, :), egoPolygon(2, :), "g", "Parent", hAxes1)

% Visualize the Trajectory reference path and trajectory
show(localPlanner, "Trajectory", "optimal", "Parent", hAxes1)
end

% Execute and Update Visualization
[isGoalReached, currentEgoState] = ...
    exampleHelperExecuteAndVisualize(currentEgoState, simStep, ...
    trajectory, previousReplanTime);
if(isGoalReached)
    break;
end

% Update the simulation step for the next iteration
simStep = simStep + 1;
pause(0.01);
end

```



At the end of this execution, the ego vehicle is at position C.

Next, configure `trajectoryOptimalFrenet` for performing a lane change maneuver from position C to position D.

Use `trajectoryOptimalFrenet` to perform Lane Change maneuver

Lane change maneuver can be performed by appropriately configuring the Lateral terminal states of the planner. This can be achieved by setting the lateral terminal state to lane width (3.6m in this example) and assuming the reference path is aligned to the center of the current ego lane.

```
% Simulate till the ego vehicle reaches position D
% Set Lane Width
laneWidth = 3.6;
% Check only for Y as there is no change in X at D
while currentEgoState(2) >= positionD(2)

    % Replan at every "replanInterval" simulation step
    if rem(simStep, replanInterval) == 0
        % Compute the replanning time
        previousReplanTime = simStep*timeResolution;
```

```

% Updating occupancy map with vehicle information
exampleHelperUpdateOccupancyMap(otherVehicles,simStep,currentEgoState);

% TerminalState settings for negotiating Lane change
localPlanner.TerminalStates.Longitudinal = 20:5:40;
localPlanner.TerminalStates.Lateral = laneWidth;
localPlanner.TerminalStates.Speed = 10;
desiredTimeBound = localPlanner.TerminalStates.Longitudinal/...
    ((currentEgoState(1,5) + localPlanner.TerminalStates.Speed)/2);
localPlanner.TerminalStates.Time = desiredTimeBound;
localPlanner.FeasibilityParameters.MaxCurvature = 0.5;
localPlanner.FeasibilityParameters.MaxAcceleration = 15;

% Generate optimal trajectory for current set of parameters
currentStateInFrenet = cart2frenet(localPlanner,[currentEgoState(1:5) 0]);
trajectory = plan(localPlanner,currentStateInFrenet);

% Visualize the ego-centric occupancy map
show(egoMap,"Parent",hAxes1);
hAxes1.Title.String = "Ego Centric Binary Occupancy Map";

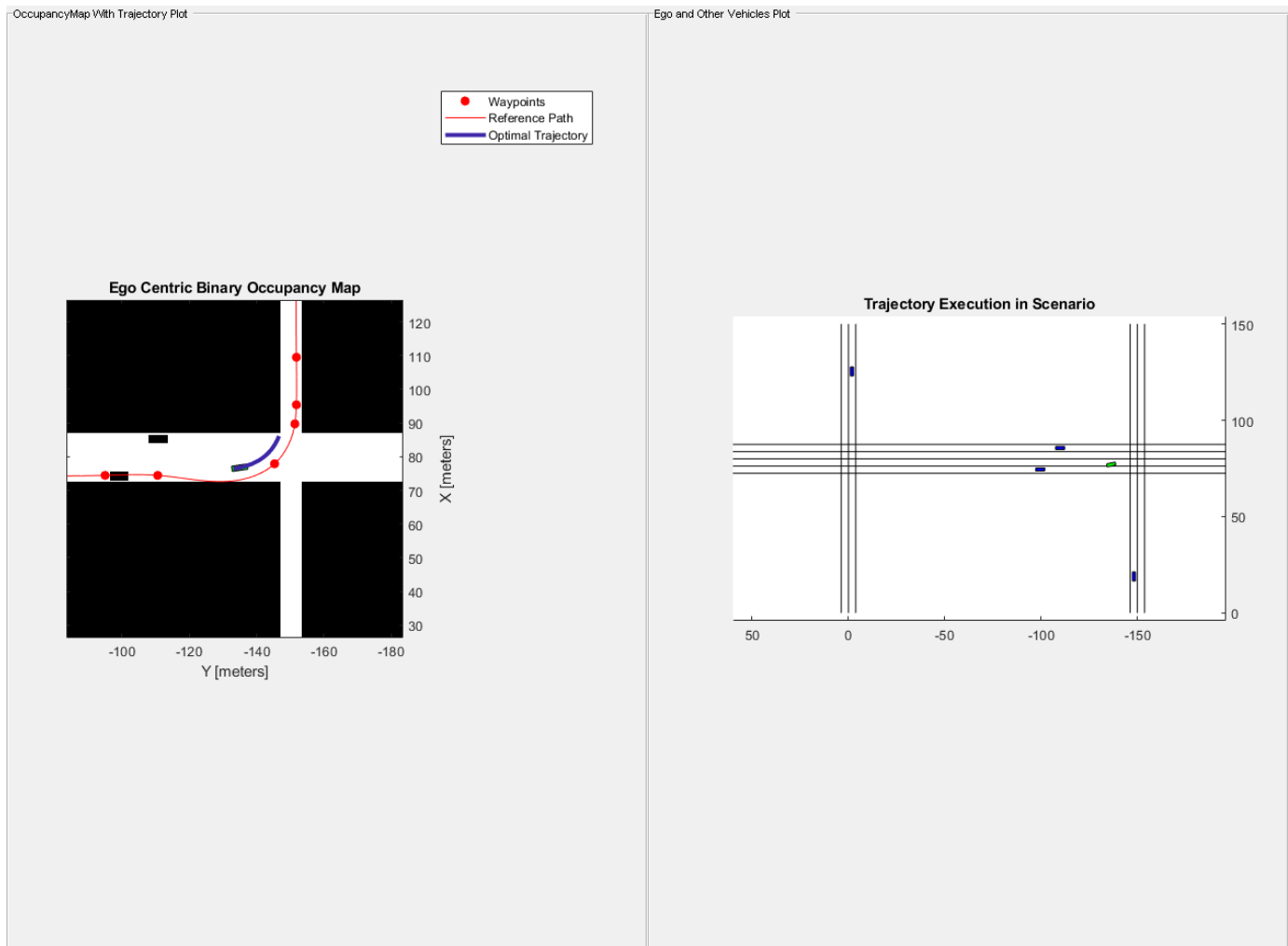
% Visualize ego vehicle on occupancy map
egoCenter = currentEgoState(1:2);
egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)), egoCenter,
hold(hAxes1,"on")
fill(egoPolygon(1, :),egoPolygon(2, :),"g","Parent",hAxes1)

% Visualize the Trajectory reference path and trajectory
show(localPlanner,"Trajectory","optimal","Parent",hAxes1)
end

% Execute and Update Visualization
[isGoalReached, currentEgoState] = ...
    exampleHelperExecuteAndVisualize(currentEgoState,simStep,...
    trajectory,previousReplanTime);
if(isGoalReached)
    break;
end

% Update the simulation step for the next iteration
simStep = simStep + 1;
pause(0.01);
end

```



Simulate ego vehicle execution to reach the goal point

The localPlanner is now configured to navigate from position D to the goal position.

```
% Simulate till the ego vehicle reaches Goal position
% Check only for X as there is no change in Y at Goal.
while currentEgoState(1) <= goalPoint(1)

    % Replan at every "replanInterval"th simulation step
    if rem(simStep, replanInterval) == 0
        % Compute the replanning time
        previousReplanTime = simStep*timeResolution;

        % Updating occupancy map with vehicle information
        exampleHelperUpdateOccupancyMap(otherVehicles, simStep, currentEgoState);
        localPlanner.TerminalStates.Longitudinal = 20;
        localPlanner.TerminalStates.Lateral = [-1 0 1];
        desiredTimeBound = localPlanner.TerminalStates.Longitudinal/...
            ((currentEgoState(1,5) + localPlanner.TerminalStates.Speed)/2);
        localPlanner.TerminalStates.Time = desiredTimeBound:0.2:desiredTimeBound+1;

        % Generate optimal trajectory for current set of parameters
```

```

currentStateInFrenet = cart2frenet(localPlanner, [currentEgoState(1:5) 0]);
trajectory = plan(localPlanner, currentStateInFrenet);

% Visualize the ego-centric occupancy map
show(egoMap,"Parent",hAxes1);
hAxes1.Title.String = "Ego Centric Binary Occupancy Map";

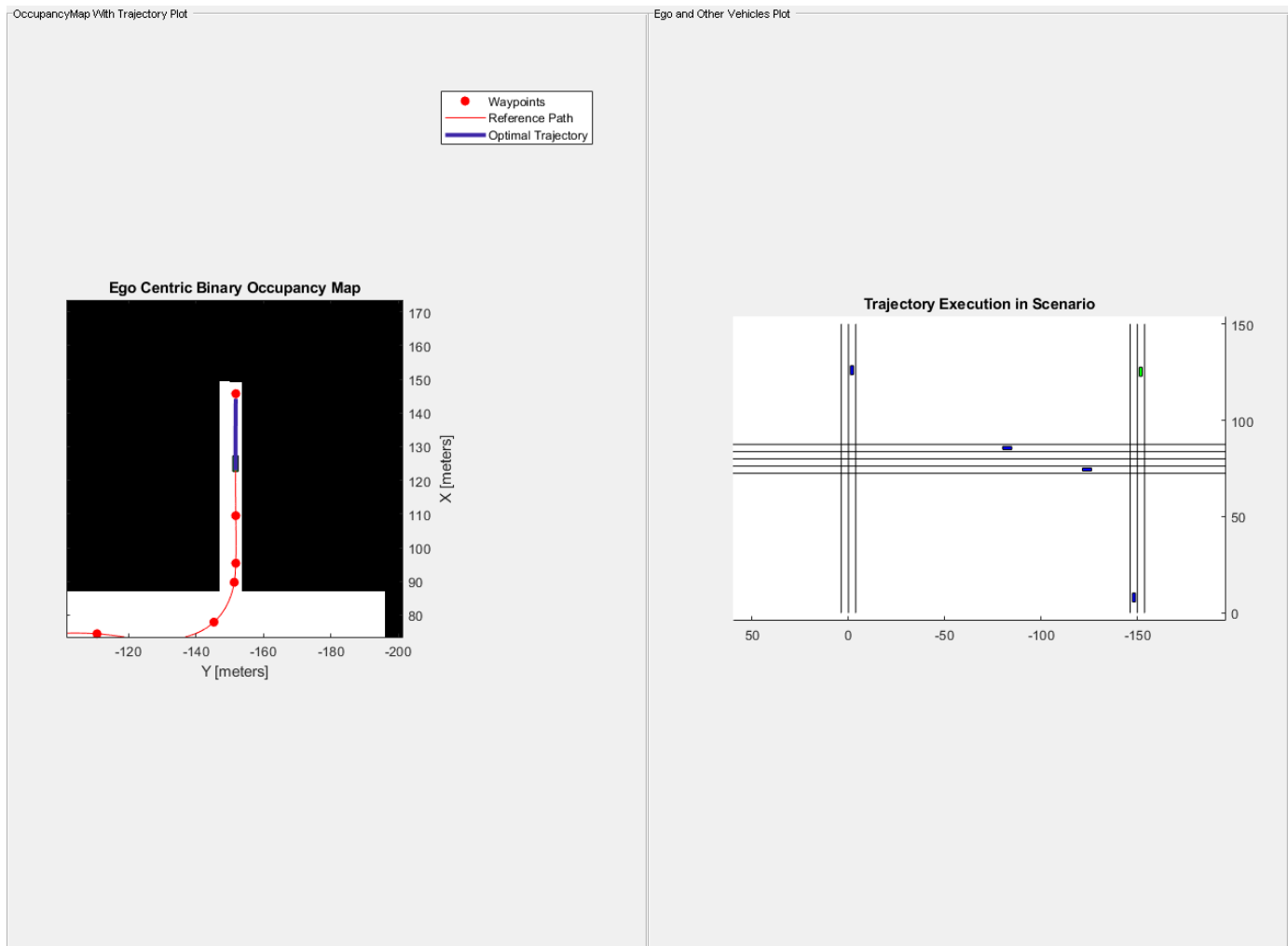
% Visualize ego vehicle on occupancy map
egoCenter = currentEgoState(1:2);
egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)), egoCenter);
hold(hAxes1,"on")
fill(egoPolygon(1, :),egoPolygon(2, :),"g","Parent",hAxes1)

% Visualize the Trajectory reference path and trajectory
show(localPlanner,"Trajectory","optimal","Parent",hAxes1)
end

% Execute and Update Visualization
[isGoalReached, currentEgoState] = ...
    exampleHelperExecuteAndVisualize(currentEgoState,simStep,...
    trajectory,previousReplanTime);
% Goal reached will be true only in this section.
if(isGoalReached)
    break;
end

% Update the simulation step for the next iteration
simStep = simStep + 1;
pause(0.01);
end

```



Log the ego vehicle positions in `egoPoses` variable that is available in the base workspace. You can playback the vehicle path in `DrivingScenario` using `exampleHelperPlayBackInDS(egoPoses)`.

```
% Clear workspace variables that were created during the example run.  
% This excludes egoPoses to allow the user to playback the simulation in DS  
exampleHelperUrbanCleanUp;
```

Conclusion

This example has explained how to perform dynamic re-planning in an urban scenario using `trajectoryOptimalFrenet`. In particular, we have learned how to use `trajectoryOptimalFrenet` to realize the following behavior.

- Adaptive Cruise Control
- Negotiating turns
- Lane Change Maneuver.

EKF-Based Landmark SLAM

This example shows how to use ekfSLAM for a reliable implementation of landmark Simultaneous Localization and Mapping (SLAM) using the Extended Kalman Filter (EKF) algorithm and maximum likelihood algorithm for data association. In this example, you create a landmark map of the immediate surroundings of a vehicle and simultaneously track the path of the vehicle. Generate a trajectory by moving the vehicle using the noisy control commands, and form the map using the landmarks it encounters along the path. Correct the vehicle trajectory and landmark estimates by observing the landmarks again.

Load Data Set

Load a modified version of the Victoria Park data set that contains the controller inputs, measurements, GPS latitude and longitude, and dead reckoning generated using the controller inputs and motion model.

```
load("victoriaParkDataset.mat","controllerInput", ...
     "measurements","gpsLatLong","deadReckoning");
```

Set Up Parameters

Specify the initial vehicle state and state covariance.

```
initialState = [gpsLatLong(1,2) gpsLatLong(1,1) deg2rad(37)]';
initialCovar = eps*eye(3);
```

Specify the process noise covariance in velocity and steering commands.

```
sigmaVelocity = 2;      % [m/s]
sigmaSteer= deg2rad(6); % [rad]
processNoise = [sigmaVelocity^2 0; 0 sigmaSteer^2];
```

Specify the measurement covariance in range and bearing.

```
sigmaRange = 1;          % [m]
sigmaBearing = deg2rad(3); % [rad]
measCovar = [sigmaRange^2 sigmaBearing^2];
```

Specify the maximum range at which to check landmarks for association.

```
maxSensorRange = 30; % [m]
```

Specify the time step size in which the vehicle moves.

```
timeStep = 0.025; % [sec]
```

Specify the thresholds for the data association function, `nav.algs.associateMaxLikelihood`. The landmark rejection threshold is the chi-square (χ^2) distribution table value for a 95% correct association.

```
landmarkRejectionThres = 5.991; % maximum distance for association
landmarkAugmentationThres = 200; % minimum distance for creation of new landmark
validationGate = [landmarkRejectionThres landmarkAugmentationThres];
```

Set a flag that determines whether to plot the map during the run.

```
plotOnTheRun = false;
```

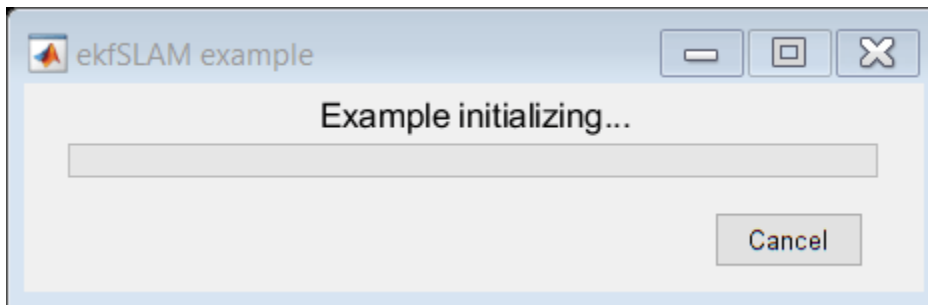
Get the number of samples from the data set.

```
numSamples = size(controllerInput,1);
```

Initialize Figure or Progress Bar

If the live plot option is enabled, create a new figure and set up handles for plotting various components. Otherwise, create a wait bar that updates with the proportion of the data set that has been executed. Set the wait bar so that clicking Cancel stops an in-progress computation. Closing the wait bar stops the execution.

```
if plotOnTheRun
    [robotHandle,covarianceHandle,sensorHandle, ...
    observationHandle,landmarkHandle, ...
    deadRecHandle,estTrajHandle] = ...
    exampleHelperInitializeVisualizationEKFSLAM(initialState,gpsLatLong);
else
    waitBarHandle = waitbar(0,'Example initializing...', ...
        'Name','ekfSLAM example', ...
        'CreateCancelBtn','setappdata(gcf,'canceling',1)');
    setappdata(waitBarHandle,'canceling',0);
end
```



Set Up ekfSLAM

Configure the ekfSLAM object using the initial state of the vehicle, initial state covariance, process noise covariance, and the motion model of the vehicle.

```
ekfSlamObj = ekfSLAM('State',initialState, ...
    'StateCovariance',initialCovar, ...
    'StateTransitionFcn',@exampleHelperVictoriaParkStateTransition);
ekfSlamObj.ProcessNoise = processNoise;
ekfSlamObj.MaxAssociationRange = maxSensorRange;
```

Main Loop

The main loop consists of these primary operations:

- **Prediction** — Predict the next state based on the control command and the current state.
- **Landmark Extraction** — Get the landmarks in the environment.
- **Correction** — Update the state and state covariance using the observed landmarks.

Prediction

In this example, the vehicle moves relative to its previous state based on the control input while the landmarks remain stationary. Thus, only the state of the vehicle is propagated. Use the motion model of the vehicle to propagate the state of the vehicle to next time step using the data set.

The `predict` method calls the function specified in the `StateTransitionFcn` property of `ekfSlamObj` to predict the state of the vehicle. The `predict` method passes the control input, other necessary inputs, and the current vehicle pose to the `StateTransitionFcn`.

```
for count = 1:numSamples
    predict(ekfSlamObj,controllerInput(count,:),timeStep);
```

Landmark Extraction

This example uses a list of observed landmarks from the data set, so you do not have to extract landmarks from the environment using a sensor. The landmarks are from a semi-circular region in front of the vehicle. The radius of the semi-circular region is defined by the maximum range of the sensor.

```
observedLandmarks = measurements{count};
```

Correction

The `ekfSLAM` object corrects the state based on a given measurement, returning a list of matched landmarks and a list of new landmarks. The `correct` method uses the function specified in the `DataAssociationFcn` property of `ekfSlamObj` to associate the observed landmarks or measurements to known landmarks. The data association function returns a list of associations and a list of new landmarks. The `correct` method uses the associations to update the belief of the vehicle and the map using the correlation between the observed landmarks and the known landmarks. Additionally, the `correct` method augments the position and covariance of any new landmark in the `State` and `StateCovariance` vectors, respectively.

If the live plot option is enabled, update the figure with the scans at the current time.

```
if ~isempty(observedLandmarks)
    correct(ekfSlamObj,observedLandmarks,measCovar,validationGate);
    % Update the corrected position of vehicle in the figure
    if plotOnTheRun
        exampleHelperUpdateScans(ekfSlamObj.State, ...
                                ekfSlamObj.MaxAssociationRange, ...
                                observedLandmarks, ...
                                sensorHandle,observationHandle);
    end
end
```

Use `drawnow` to update the figure with the current position of the vehicle and all the known landmarks and their covariance.

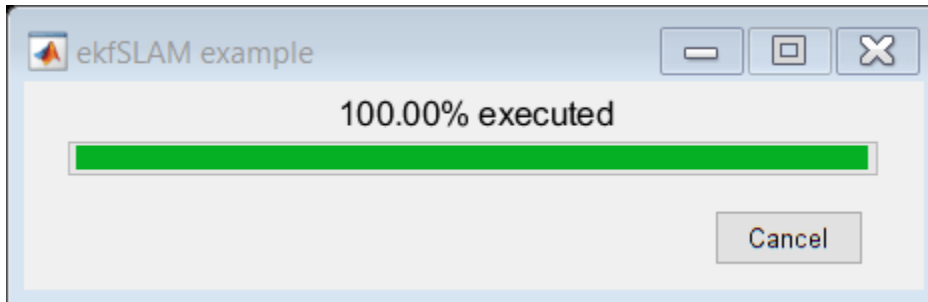
If the live plot is not enabled and the wait bar is active, then update the fraction of the data set executed.

```
if plotOnTheRun
    addpoints(deadRecHandle,deadReckoning(count,1),deadReckoning(count,2));
    addpoints(estTrajHandle,ekfSlamObj.State(1),ekfSlamObj.State(2));
    exampleHelperUpdateRobotAndLandmarks(ekfSlamObj.State,ekfSlamObj.StateCovariance, ...
        robotHandle,covarianceHandle,landmarkHandle);
```

```

        drawnow limitrate
    else
        % update the wait bar
        waitbar(count/numSamples,waitBarHandle, ...
            sprintf("%2.2f%% executed",count/numSamples*100));
        % Check for clicked Cancel button
        if getappdata(waitBarHandle,"canceling")
            break
        end
    end
end

```



```

end

```

Use the `delete` function to close the wait bar once all the computations are complete.

```

if ~plotOnTheRun
    delete(waitBarHandle);
end

```

Visualize Map

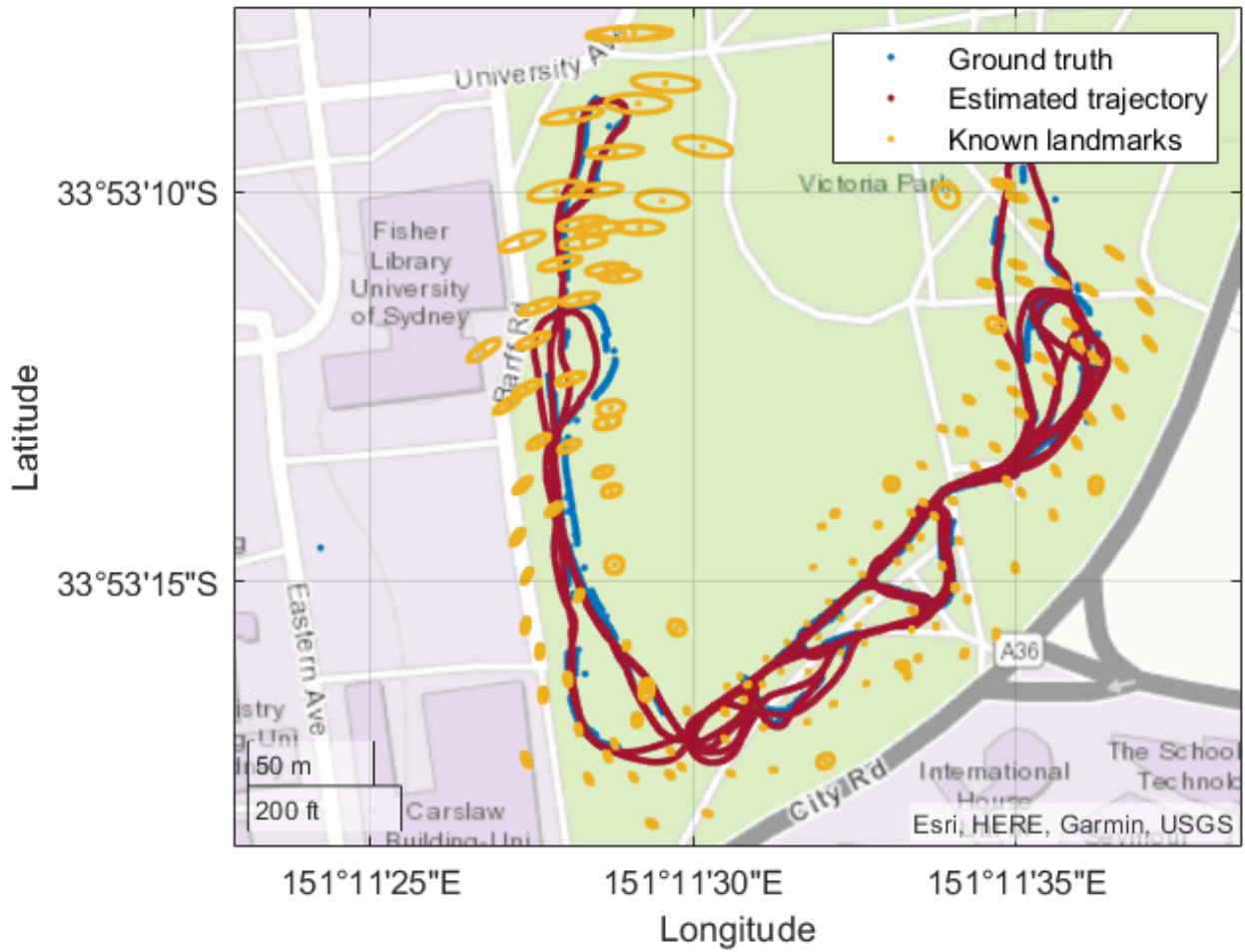
Create a geographic plot with the ground truth from the GPS data. Plot the vehicle trajectory, along with all detected landmarks and their associated covariance, on the same geographic plot.

```

% get the corrected and predicted poses
[corrPoses,predPoses] = poseHistory(ekfSlamObj);

% show the map
exampleHelperShowMap(ekfSlamObj.State,ekfSlamObj.StateCovariance, ...
    gpsLatLong,corrPoses,predPoses);

```



Reverse-Capable Motion Planning for Tractor-Trailer Model Using `plannerControlRRT`

This example shows how to find global path-planning solutions for systems with complex kinematics using the kinematics-based planner, `plannerControlRRT`. The example is organized into three primary sections:

- 1 Introduction to Kino-Dynamic Planning.
- 2 Adapting `plannerControlRRT` to a Tractor-Trailer System.
- 3 Finding Collision-Free Trajectories Using a Kinematic System.

Introduction to Kino-Dynamic Planning

Planner Description

Conventional geometric planners, such as RRT, RRT*, and `hybridA*`, are fast and extensible algorithms capable of finding complete and optimal solutions to a wide variety of planning problems. One trade-off, however, is that they make assumptions about the planning space that may not hold true for a real-world system. Conventional geometric planners assume that any two states in a space can be connected by a trajectory with no residual error, and that the paths returned by the geometric planner can be tracked by the physical system.

For systems with complex kinematics, or those that do not have easily determined closed-form solutions for connecting states, use kino-dynamic planners, such as `plannerControlRRT`. Kino-dynamic planners trade completeness for planning flexibility, and leverage a system's own kinematic model and controller to generate feasible and trackable trajectories.

To plan for a given system, the `plannerControlRRT` feature requires this information:

- A utility to sample states in the planning space.
- A metric to estimate the cost of connecting two states.
- A mechanism to deterministically propagate the system from one state toward another.
- A utility to determine whether a state has reached the goal.

This example demonstrates how to formulate these different utilities for a tractor-trailer system and adapt them into the planning infrastructure. This preview shows the result:

```
% Attempt to generate MEX to accelerate collision checking
genFastCC
```

```
Code generation successful.
```

```
% load preplannedScenarios
parkingScenario = load("parkingScenario.mat");
planner = parkingScenario.planner;
start    = parkingScenario.start;
goal     = parkingScenario.goal;
```

```
% Display the scenario and play back the trajectory
show(planner.StatePropagator.StateValidator);
axis equal
hold on
```

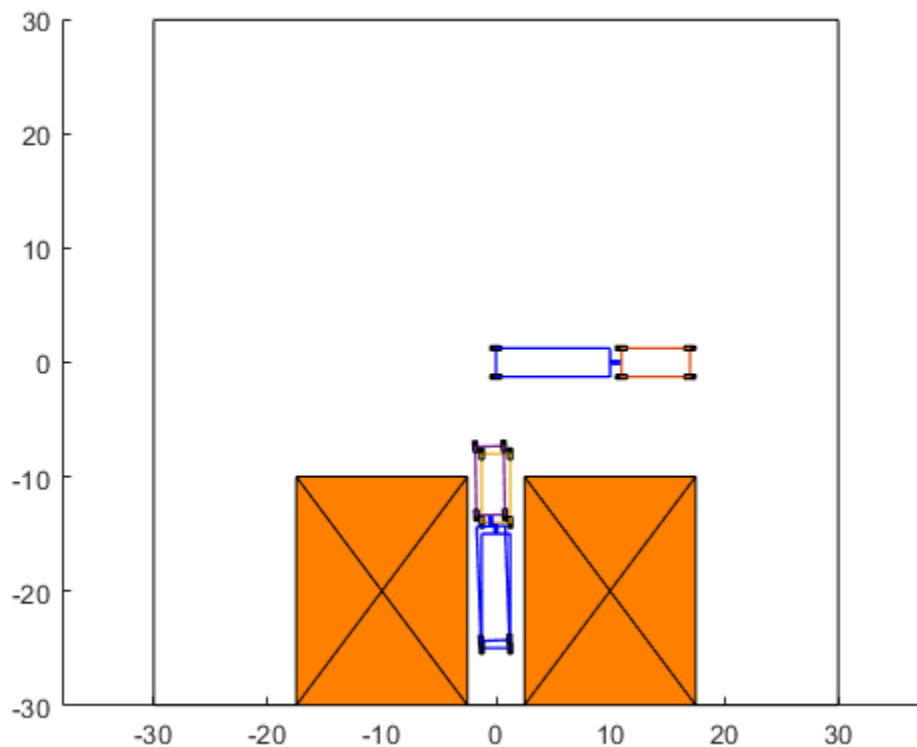
```

trajectory = plan(planner,start,goal);
demoParams = trajectory.StatePropagator.StateSpace.TruckParams;
exampleHelperPlotTruck(demoParams,start); % Display start config
exampleHelperPlotTruck(demoParams,goal); % Display goal config

% Plan path using previously-built utilities
rng(1)
trajectory = plan(planner,start,goal);

% Animate the trajectory
exampleHelperPlotTruck(trajectory);
hold off

```



Kino-Dynamic Planning Algorithm

If you are already familiar with geometric planners, an iteration of the `plannerControlRRT` search algorithm should look familiar, but with a few extra steps:

- 1 *Sample* a target state.
- 2 Find the approximate *nearest neighbor* to the target state in the search tree.
- 3 *Generate* a control input (or reference value) and control duration that drives the system toward the target.
- 4 *Propagate* (extend) the system towards the sample while *validating* intermediate states along the propagated trajectory. Return the valid sequence of states, controls, and durations.
- 5 *Check* whether any returned states have reached the goal.

- 6 If the goal has been found, *exit*. Otherwise *add* the final state, initial control, cumulative duration, and target to the tree.
- 7 (Optional) Sample a goal-configuration and control and repeat steps 4,5, and 6 the desired number of times.

Components Required by `plannerControlRRT` Framework

To adapt the planner to a specific problem, you must implement custom `nav.StatePropagator` and `nav.StateSpace` classes. The section below maps the steps described in the Kino-Dynamic Planning Algorithm on page 1-0 section to the class and method responsible:

- 1 $q_{tgt} = \text{sampleUniform}(\textit{space})$
- 2 $[\textit{dist}, \textit{err}] = \text{distance}(\textit{propagator}, Q_{\textit{tree}}, q_{\textit{tgt}})$
- 3 $[u, N_{\textit{step}}] = \text{sampleControl}(\textit{propagator}, q_0, u_0, q_{\textit{tgt}})$
- 4 $[Q_{1:n}, U_{1:n}, n_{\textit{step}1:n}]_{\textit{valid}} = \text{propagateWhileValid}(\textit{propagator}, q_0, u_0, q_{\textit{tgt}}, N_{\textit{step}})$
- 5 `plannerControlRRT` checks whether any returned states have reached the goal.
- 6 By default, exiting planning or adding to the search tree is performed by the `distance` function of the propagator. You can override it by supplying the planner with a function handle during construction.
- 7 By default, each time the planner successfully adds a node to the tree, it will attempt to propagate this newly added state towards the goal-configuration. This goal-propagation can occur multiple times in a row, or disabled entirely, by modifying the `NumGoalExtension` property of the planner.

For more detailed function syntax, argument requirements, and examples, see `plannerControlRRT`, `nav.StateSpace`, and `nav.StatePropagator`.

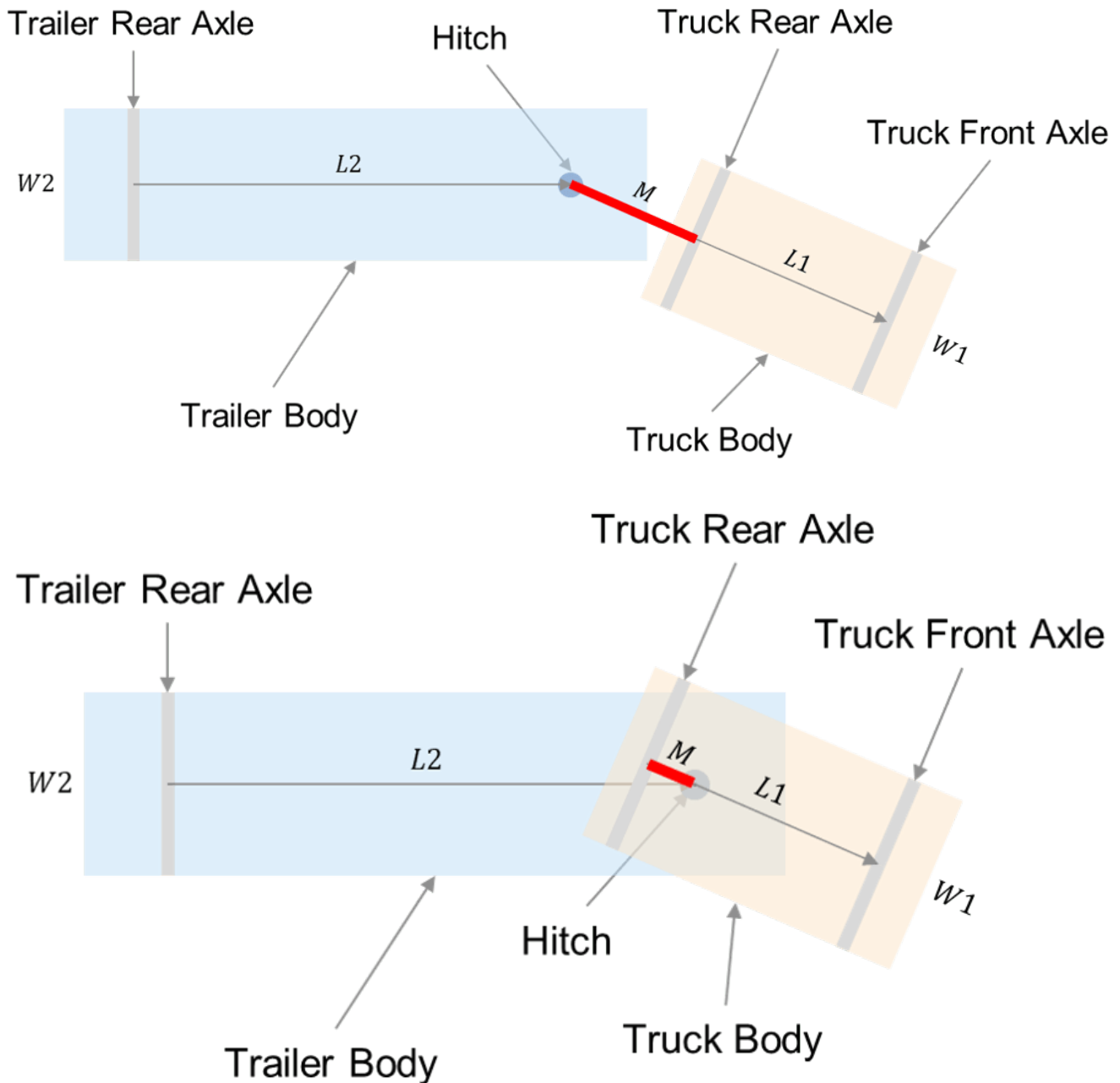
Adapting `plannerControlRRT` to a Truck-Trailer System

This section demonstrates how to formulate, model, and control a two-body truck-trailer system. This system is inherently unstable during reverse motion, so paths returned by conventional geometric planners are unlikely to produce good results during path following. To accurately model this system, you must:

- Define a state transition function.
- Define the geometric model parameters.
- Create a control law attuned to the parameterized model.
- Create a distance heuristic attuned to the parameterized model.
- Package the kinematic model, distance metric, and control law inside a custom state space and state propagator.
- Plan a path for several problem scenarios.

Define State Transition Function

This tractor-trailer system is comprised of two rigid bodies connected at a *kingpin* hitch, which is modeled as a revolute joint. The trailer body makes contact with the ground at the rear axle, and the front of the trailer is supported by the hitch, located somewhere along the axial centerline of the truck.



Trailing Configuration (left, $M < 0$), OverCenter Configuration (right, $M > 0$)

Use the geometric configuration for a two-body truck-trailer system from the Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC example, expressed as:

$$q_{\text{sys}} = [x_2 \ y_2 \ \theta_2 \ \beta],$$

where:

- x_2 — Global x -position of the center of the trailer rear axle.

- y_2 — Global y -position of the center of the trailer rear axle.
- θ_2 — Global angle of the trailer orientation, where θ is east.
- β — Truck Orientation with respect to the trailer, where θ is aligned.

For planning purposes, append a direction flag, v_{sign} , and the total distance traveled from the start configuration, s_{tot} to the state vector, for the final state notation:

$$q = [x_2 \ y_2 \ \theta_2 \ \beta \ v_{\text{sign}} \ s_{\text{tot}}]$$

- v_{sign} — Indicates the desired control mode (forward or reverse) or a required velocity, ensuring the system propagates toward the goal occur in a desired direction.
- s_{tot} — Modifies the behavior of the propagator's distance function of the propagator. You can modify the propagator's `TravelBias` property to change the frequency with which the nearest-neighbor comparison includes or excludes the root-to-node cost. Values closer to 0 result in a faster search of the planning space, at the expense of more locally optimal connections being made. Values closer to 1 can improve planner solutions, but increase planning time.

This state transition function is an extension of `TruckTrailerStateFcn`. For a full derivation of the state transition function, see `exampleHelperStateDerivative`, which is a simplification of this N-trailer model [1] on page 1-0 :

$$\dot{q}(L_1, L_2, M, q, u_{\text{app}}) = [\dot{x}_2 \ \dot{y}_2 \ \dot{\theta}_2 \ \dot{\beta} \ 0 \ v]$$

For simplicity, this model treats $u_{\text{app}} = [v, \alpha_{\text{steer}}]$ as the instantaneous velocity and steering angle command applied to the system, generated by a higher-level controller. For more information, see `Create Control Laws for Stable Forward and Reverse Motion` on page 1-0 .

Define Geometric Model Parameters and Demonstrate Instability

The kinematic characteristics of this vehicle are heavily influenced by the choice of model parameters. Create a structure that describes a tractor-trailer system,

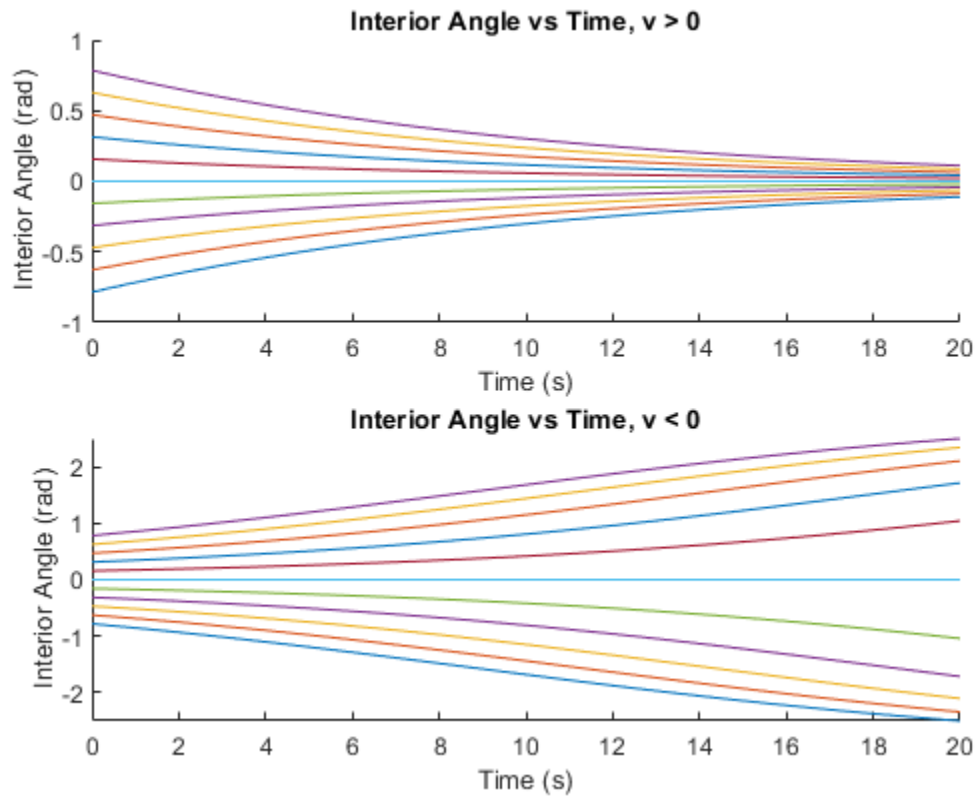
```
truckParams = struct;
truckParams.L1 = 6;      % Truck body length
truckParams.L2 = 10;    % Trailer length
truckParams.M = 1;      % Distance between hitch and truck rear axle along truck body +X-axis
```

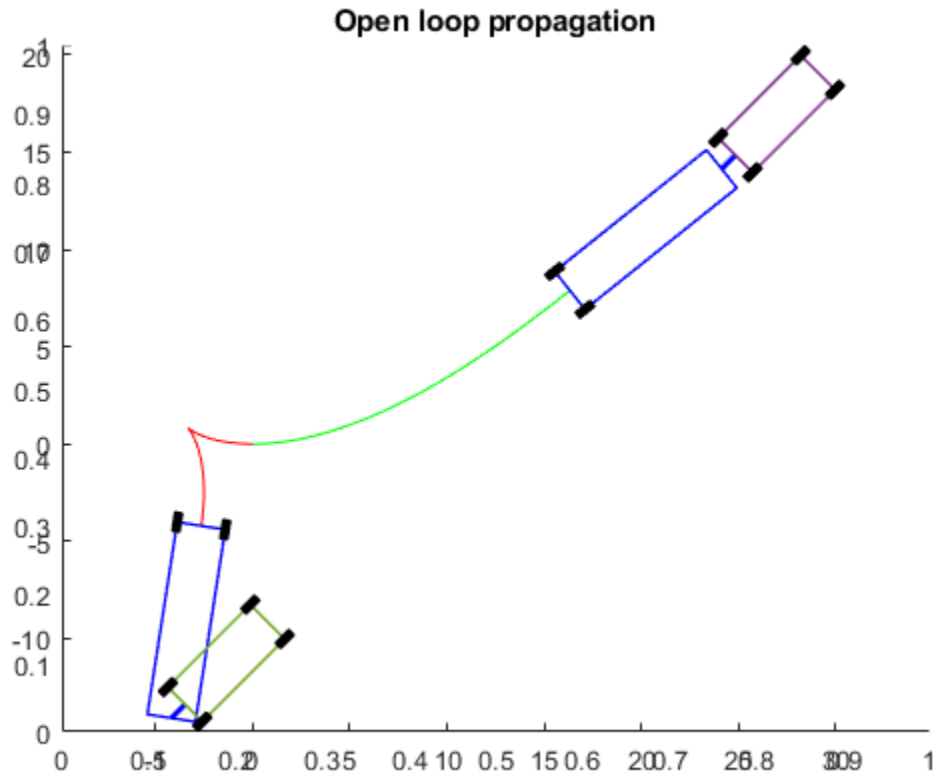
Define properties related to visualization and collision-checking.

```
truckParams.W1 = 2.5;    % Truck width
truckParams.W2 = 2.5;    % Trailer width
truckParams.Lwheel = 1;  % Wheel length
truckParams.Wwheel = 0.4; % Wheel width
```

Using the `exampleHelperShowOpenLoopDynamics` function and specified model parameters, propagate the system state using its open-loop dynamics to demonstrate that the interior angle is self-stabilizing during forward motion, and unstable during reverse motion for any non-zero β_0 .

```
exampleHelperShowOpenLoopDynamics(truckParams);
```





Create Control Laws for Stable Forward and Reverse Motion

This example employs a multi-layer, mode-switching controller to control the truck-trailer model. At the top level, a pure pursuit controller calculates a reference point, $P = [X_p, Y_p]$, between a current pose, q_i , and goal state, q_{tgt} . The controller has two modes, for forward and reverse. For information on the basis for this control law, see [1] on page 1-0 . For forward motion, the controller calculates a steering angle that, when held constant, drives the rear axle of the truck along an arc that intersects the reference point:

$$q_{1i} = [x_1, y_1, \theta_1]_i$$

$$\theta_{err} = \text{atan2}(X_p - x_1, Y_p - y_1) - \theta_1$$

$$\alpha_{fwd} = f(\theta_{err})$$

For reverse motion, the rear axle of the trailer becomes the controlled state, $q_2 = [x_2, y_2, \theta_2]$. Furthermore, because the system is inherently unstable during reverse motion, the control law treats the steering angle returned by the top-level controller as a reference value to a gain-scheduled LQ controller, which seeks to stabilize the interior angle of the vehicle:

$$\theta_{err} = \text{atan2}(X_p' - x_2, Y_p' - y_2) - \theta_2$$

$$\beta_d = -\text{atan}\left(\frac{2L_2 \sin(\theta_{err})}{R}\right)$$

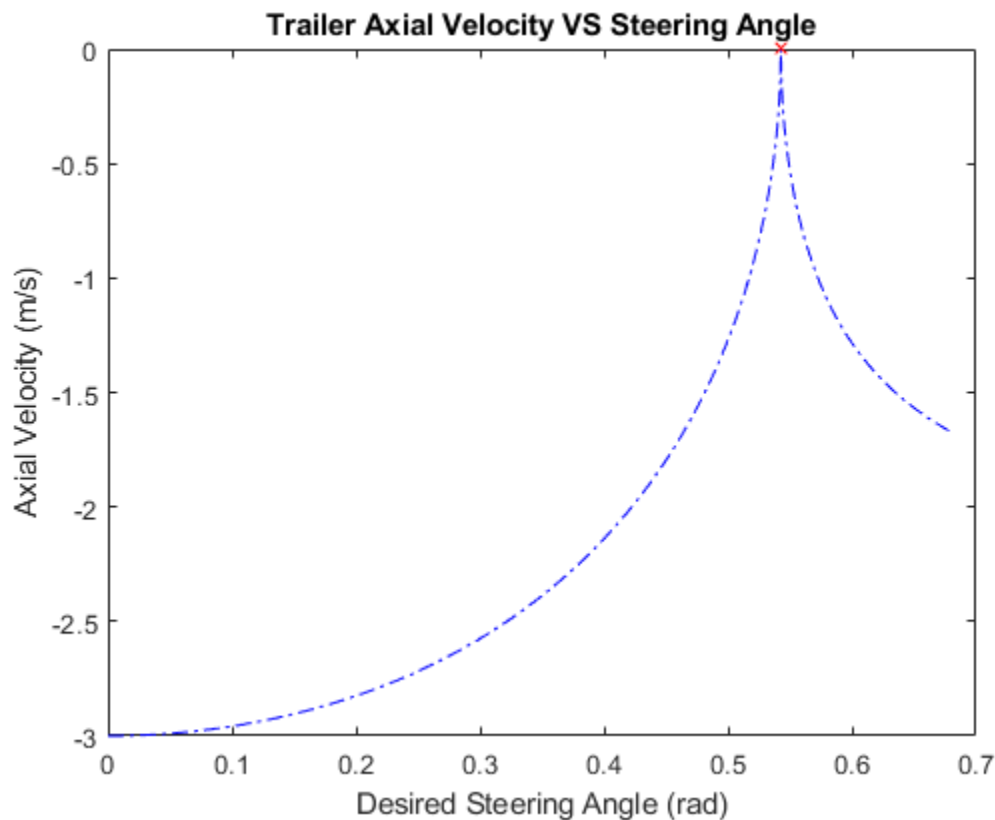
$$\beta_{\text{ref}} = \beta_d + K_p(\beta_d - \beta)$$

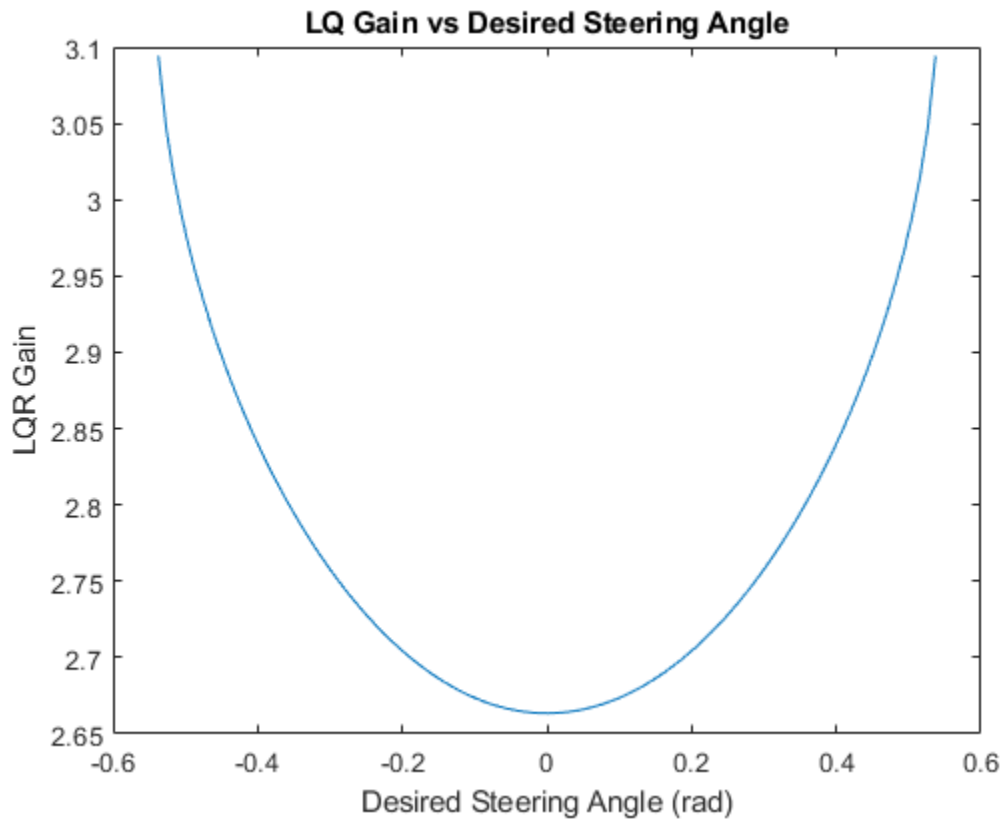
LQ Feedback Stabilization

During reverse motion, you can add a gain to the steering angle that drives the interior angle to an equilibrium point. Use the LQ controller to calculate the desired steering angle and feedback gains. The gains are the optimal solution to the Algebraic Ricatti equation, stored as a lookup table dependent on the desired steering angle. For more information about the derivation of this feedback controller, see `exampleHelperCalculateLQGains`:

```
% Define Q/R weight matrices for LQR controller
Q = 10; % Weight driving betaDot -> 0
R = 1; % Weight minimizing steering angle command

% Derive geometric steering limits and solve for LQR feedback gains
[alphaLimit, ... % Max steady-state steering angle before jack-knife
 betaLimit, ... % Max interior angle before jack-knife
 alphaDesiredEq, ... % Sampled angles from stable alpha domain
 alphaGain ... % LQ gain corresponding to desired alpha
] = exampleHelperCalculateLQGains(truckParams,Q,R);
```





```
% Add limits to the truck geometry
truckParams.AlphaLimit = alphaLimit;
truckParams.BetaLimit = betaLimit;
```

Specify lookahead distances for the pure pursuit controller, `exampleHelperPurePursuitGetSteering`, and rate and time-span information for our model simulator, `exampleHelperPropagateTruck`. Because reverse motion is unstable, and you have only indirect control over the interior angle, specify a larger lookahead distance in reverse. This gives the system more time to minimize the virtual steering error while maintaining a stable interior angle.

```
% Select forward and reverse lookahead distances. For this example,
% use a reverse lookahead distance twice as long as the forward
% lookahead distance, which itself is slightly longer than the
% wheelbase. You can tune these parameters can be tuned for
% improved performance for a given geometry.
rFWD = truckParams.L1*1.2;
rREV = rFWD*2;
```

```
% Define parameters for the fixed-rate propagator and add them to the
% control structure
controlParams.MaxVelocity = 3; % m/s
controlParams.StepSize = .1; % s
controlParams.MaxNumSteps = 50; % Max number of steps per propagation
```

Create a structure of the control-related information.

```
% Store gains and control parameters in a struct
controlParams = struct(...
```

```

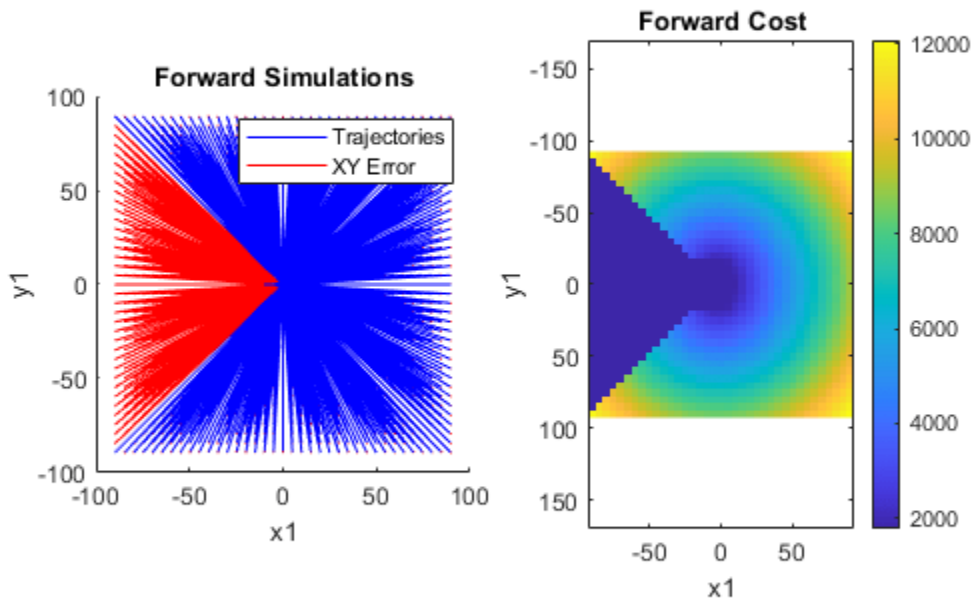
'MaxSteer',alphaLimit, ...           % (rad)
'Gains',alphaGain, ...               % ( )
'AlphaPoints',alphaDesiredEq, ...    % (rad)
'ForwardLookahead',rFWD, ...         % (m)
'ReverseLookahead',rREV, ...         % (m)
'MaxVelocity', 3, ...                % (m/s)
'StepSize', 0.1, ...                 % (s)
'MaxNumSteps', 200 ...               % ( )
);

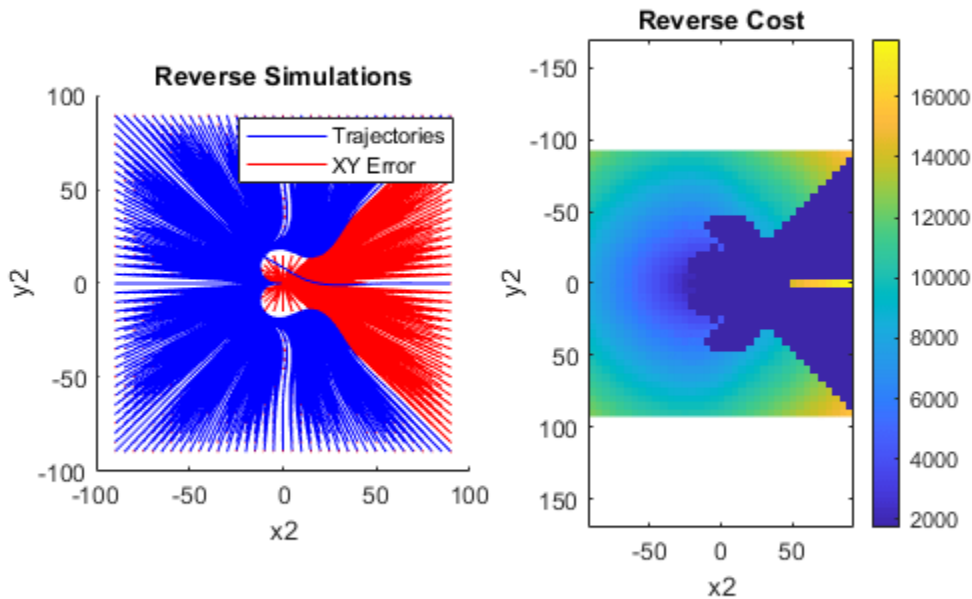
```

Define a Distance Heuristic

Define a distance heuristic to approximate the cost between different configurations in state-space of the system. The state propagator uses this when the planner attempts to find the nearest node in the tree to a sampled state. Calculate the cost offline using the `exampleHelperCalculateDistanceMetric` function, which stores it in a lookup table.

```
distanceHeuristic = exampleHelperCalculateDistanceMetric(truckParams,controlParams);
```





Adapt System for Use With Kinematic Planner

Adapt the system into a framework usable by the kinematic path-planner, `plannerControlRRT`. Create 3 custom classes: `exampleHelperTruckStateSpace`, `exampleHelperTruckPropagator`, and `exampleHelperTruckValidator`, which inherit from the `nav.StateSpace`, `nav.StatePropagator`, and `nav.StateValidator` subclasses, respectively.

Initialize State Space

The state-space object is primarily responsible for:

- Sampling random states for the planner.
- Defining and enforcing state limits within a trajectory.

Define xy -limits for the searchable region. Initialize the state-space using these limits, and the β -limits determined by the geometric properties of the truck trailer.

Because the kinematic planner uses the distance function of the state-propagator for its nearest-neighbor search, leave the state-space distance method undefined. Leave the `interpolate` and `sampleGaussian` methods of the state-space similarly undefined.

```
% Define limits of searchable region
xyLimits = [-60 60; -40 40];

% Construct our state-space
stateSpace = exampleHelperTruckStateSpace(xyLimits, truckParams);
```


Initialize State Validator

In this example, the state validator uses oriented bounding boxes (OBBs) to verify whether the vehicle is in collision with the environment. The truck is represented by two rectangles with poses defined by the state and the vehicle geometry of the truck contained in the state-space object. The environment is represented by a list of rectangles with their locations, orientations, and dimensions specified in a 1-by-N structure. Load a MAT file containing the environment into the workspace, and use the obstacles and state-space object to construct the state validator.

```
% Load a set of obstacles
load("Slalom.mat");

% Construct the state validator using the state space and list of obstacles
validator = exampleHelperTruckValidator(stateSpace,obstacles);
```

Initialize State Propagator

In kinematic planning, the state-propagator object is responsible for:

- Estimating the distance between states.
- Sampling initial controls to propagate over a control-interval, or capturing control inputs as reference values for lower-level controllers during propagation.
- Propagating the system towards a target, and returning the valid portion of the trajectory to the planner.

For best results, the distance function should estimate the cost of the trajectory generated when propagating between states.

```
% Construct the state propagator using the state validator, control parameters,
% and distance lookup table
propagator = exampleHelperTruckPropagator(validator,controlParams,distanceHeuristic);
```

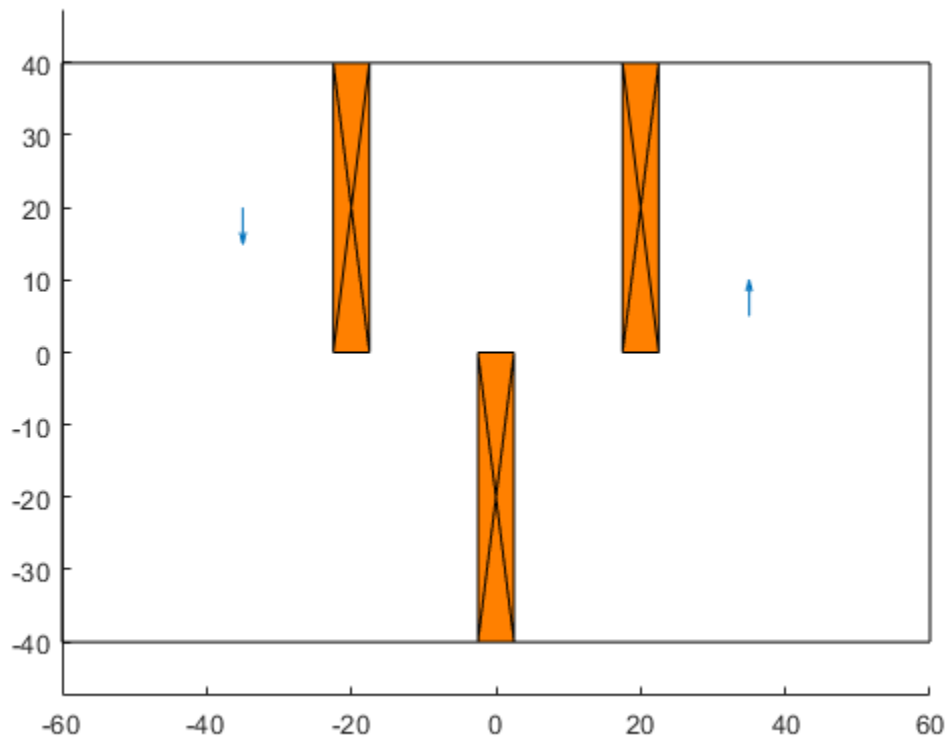
Construct Planner and Plan Path

Construct a kinematic path-planner that uses the state propagator to search for a path between the two configurations.

```
% Define start configuration
start = [35 5 pi/2 0 0 0];

% Define the goal configuration such that the truck must reverse into
% position.
goal = [-35 20 -pi/2 0 -1 nan];

% Display the problem
figure
show(validator)
hold on
configs = [start; goal];
quiver(configs(:,1),configs(:,2),cos(configs(:,3)),sin(configs(:,3)),.1)
```



```
% Define a function to check if planner has reached the true goal state
goalFcn = @(planner,q,qTgt)exampleHelperGoalReachedFunction(goal,planner,q,qTgt);
```

```
% Construct planner
```

```
planner = plannerControlRRT(propagator,GoalReachedFcn=goalFcn,MaxNumIteration=30000);
planner.NumGoalExtension = 3;
planner.MaxNumTreeNode = 30000;
planner.GoalBias = .25;
```

```
% Search for the path
```

```
rng(0)
[trajectory,treeInfo] = plan(planner,start,goal)
```

```
trajectory =
```

```
navPathControl with properties:
```

```
StatePropagator: [1x1 exampleHelperTruckPropagator]
States: [23x6 double]
Controls: [22x4 double]
Durations: [22x1 double]
TargetStates: [22x6 double]
NumStates: 23
NumSegments: 22
```

```
treeInfo = struct with fields:
```

```
IsPathFound: 1
ExitFlag: 1
```

```

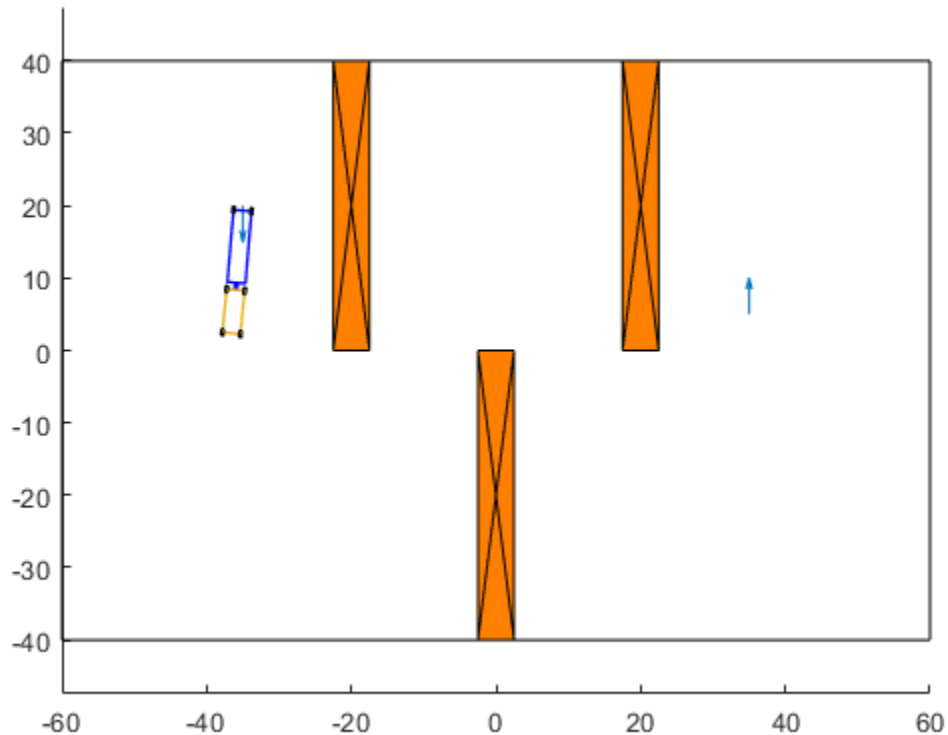
NumTreeNode: 2201
NumIteration: 1196
PlanningTime: 28.0091
TreeInfo: [6x6602 double]

```

```

% Visualize path and waypoints
exampleHelperPlotTruck(trajjectory);
hold off

```



Rather than producing optimal solutions, like those generated in the Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC example, the advantage of this planner lies in its ability to find viable trajectories for a wide variety of problems. In the scenario below, for instance, the generated path can be used as is, or as an initial guess for an MPC solver, helping to avoid local minima in the nonconvex problem space:

```

% Load scenario obstacles
nonConvexProblem = load("NonConvex.mat");

% Update validator and state space
validator.Obstacles = nonConvexProblem.obstacles;
validator.StateSpace.StateBounds(1:2,:) = [-100 100; -40 40];
figure
show(validator)

% Define start and goal
start = [10 0 0 0 NaN 0];

```

```
goal = [-10 0 pi 0 -1 0];

% Update goal reached function
goalFcn = @(planner,q,qTgt)exampleHelperGoalReachedFunction(goal,planner,q,qTgt);
planner.GoalReachedFcn = goalFcn;

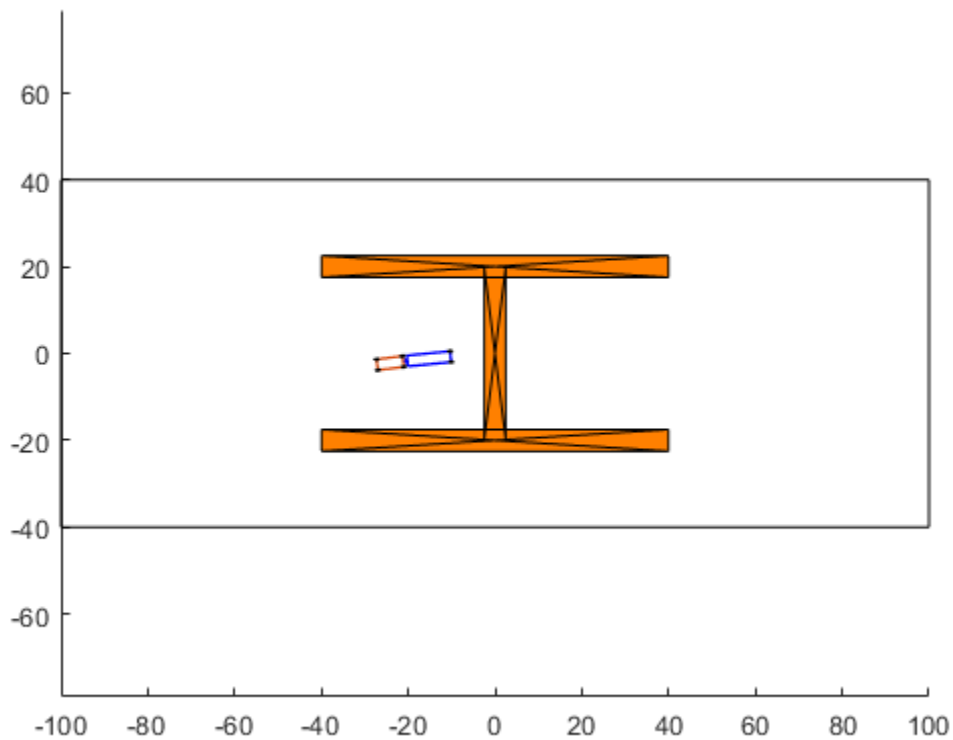
% Turn off optional every-step goal propagation
planner.NumGoalExtension = 0;

% Increase goal sampling frequency
planner.GoalBias = .25;

% Balanced search vs path optimality
propagator.TravelBias = .5;

% Plan path
rng(0)
trajectory = plan(planner,start,goal);

% Visualize result
exampleHelperPlotTruck(trajectory);
```



References

[1] Holmer, Olov. "Motion Planning for a Reversing Full-Scale Truck and Trailer System". M.S. thesis, Linköping University, 2016.

Simulate INS Block

In this example, you simulate an INS block by using the pose information of a vehicle undertaking a left-turn trajectory.

Load Vehicle Trajectory Data

First, you load the trajectory information of the vehicle to the workspace.

```
load leftTurnTrajectory.mat
```

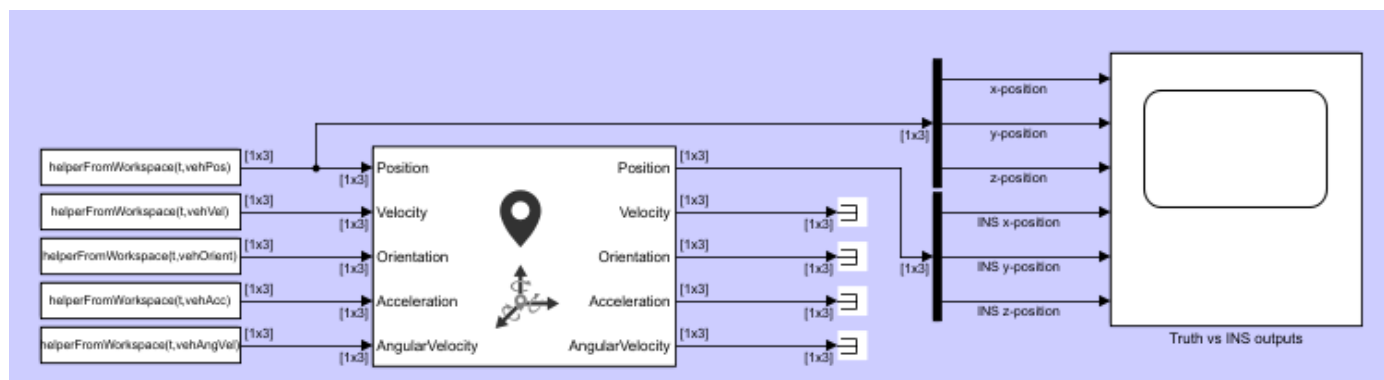
You notice that seven new variables appear in MATLAB workspace.

- `dt` — The time step size of 0.4 seconds.
- `t` — The total time span of 7.88 seconds.
- `vehPos`, `vehVel`, `vehAcc`, `vehOrient`, `vehAngVel` — The history of position, velocity, acceleration, orientation, and angular velocity, each specified as a 198-by-3 matrix, where 198 is the total number of steps.

Open Simulink Model

Next, you open the Simulink model.

```
open simulateINS.slx
```



The model contains three parts: the data importing part, the INS block, and the scope block to compare the true positions with the INS outputs.

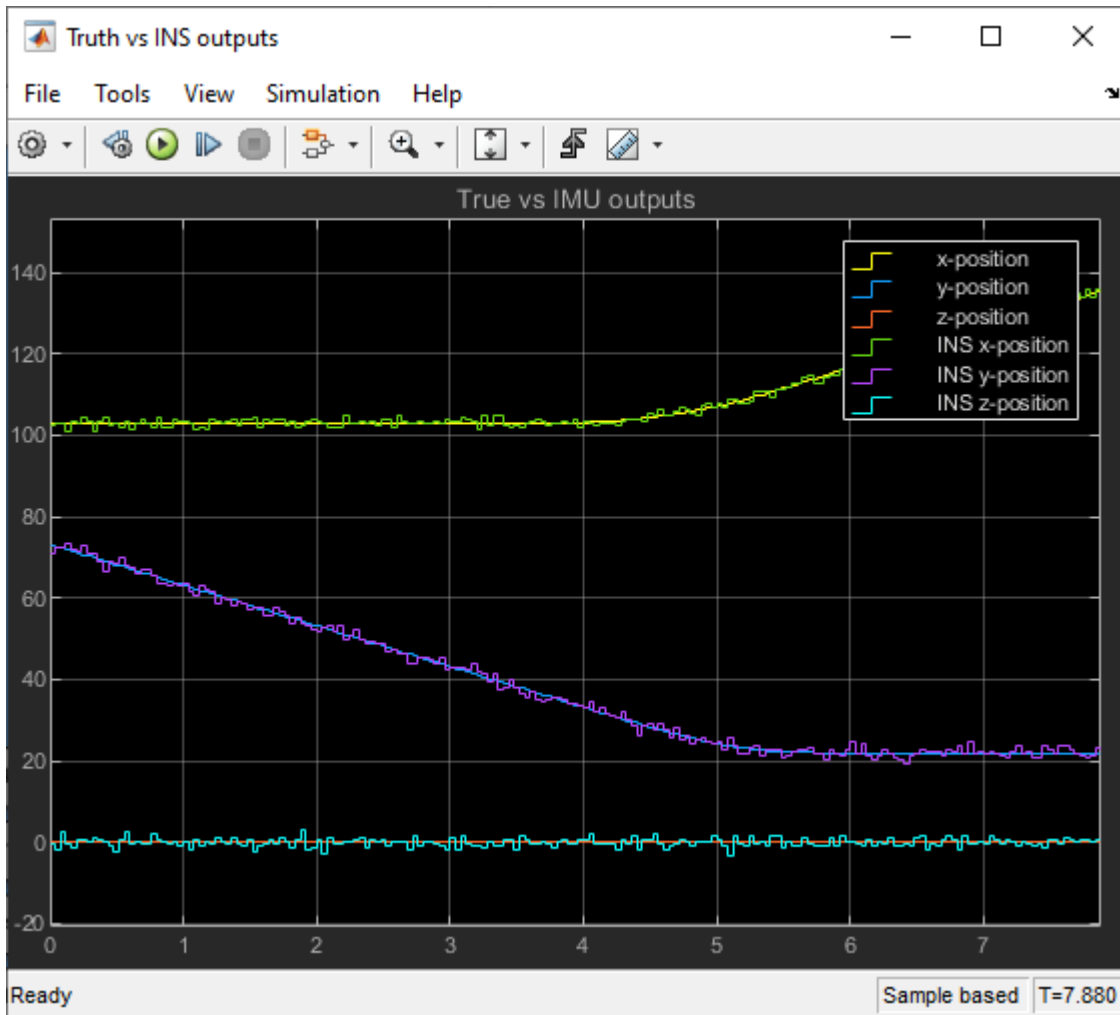
The data importing part imports the vehicle trajectory data into Simulink using the From Workspace block. You use a helper function `helperFromWorkspace`, attached in the example folder, to convert the trajectory data into a structure format required by the From Workspace block.

Run the Model

Run the Simulink model.

```
results = sim('simulateINS');
```

Click on the scope block and see the results. The INS block position outputs closely follow the truth with the addition of noise.



Object Tracking and Motion Planning Using Frenet Reference Path

This example shows you how to dynamically replan the motion of an autonomous vehicle based on the estimate of the surrounding environment. You use a Frenet reference path and a joint probabilistic data association (JPDA) tracker to estimate and predict the motion of other vehicles on the highway. Compared to the “Highway Trajectory Planning Using Frenet Reference Path” on page 1-397 example, you use these estimated trajectories from the multi-object tracker in this example instead of ground truth for motion planning.

Introduction

Dynamic replanning for autonomous vehicles is typically done with a local motion planner. The local motion planner is responsible for generating optimal trajectory based on a global plan and real-time information about the surrounding environment. The global plan for highway trajectory planning can be described as a pre-generated coordinate list of the highway centerline. The surrounding environment can be described mainly in two ways:

- 1 Discrete set of objects in the surrounding environment with defined geometries.
- 2 Discretized grid with estimates about free and occupied regions in the surrounding environment.

In the presence of dynamic obstacles, a local motion planner also requires predictions about the surroundings to assess the validity of planned trajectories. In this example, you represent the surrounding environment using the *discrete set of objects* approach. For an example using discretized grid, refer to the “Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map” (Sensor Fusion and Tracking Toolbox) example.

Object State Transition and Measurement Modeling

The object list and their future predictions for motion planning are typically estimated by a multi-object tracker. The multi-object tracker accepts data from sensors and estimates the list of objects. In the tracking community, this list of objects is often termed as *track list*.

In this example, you use radar and camera sensors and estimate the track list using a JPDA multi-object tracker. The first step towards using any multi-object tracker is defining the object state, how the state evolves with time (state transition model) and how the sensor perceives it (measurement model). Common state transition models include constant-velocity model, constant-acceleration model etc. However, in the presence of map information, road network can be integrated into the motion model. In this example, you use a Frenet coordinate system to describe the object state at any given time step, k .

$$x_k = [s_k \dot{s}_k \ d_k \ \dot{d}_k]$$

where s_k and d_k represents the distance of the object along and perpendicular to highway centerline, respectively. You use a constant-speed state transition model to describe the object motion along the highway and a decaying-speed model to describe the motion perpendicular to the highway centerline. This decaying speed model allows you to represent lane change maneuvers by other vehicles on the highway.

$$\begin{bmatrix} s_{k+1} \\ \dot{s}_{k+1} \\ d_{k+1} \\ \dot{d}_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \tau \left(1 - e^{-\frac{\Delta T}{\tau}}\right) \\ 0 & 0 & 0 & e^{-\frac{\Delta T}{\tau}} \end{bmatrix} \begin{bmatrix} s_k \\ \dot{s}_k \\ d_k \\ \dot{d}_k \end{bmatrix} + \begin{bmatrix} \frac{\Delta T^2}{2} & 0 \\ \Delta T & 0 \\ 0 & \frac{\Delta T^2}{2} \\ 0 & \Delta T \end{bmatrix} \begin{bmatrix} w_s \\ w_d \end{bmatrix}$$

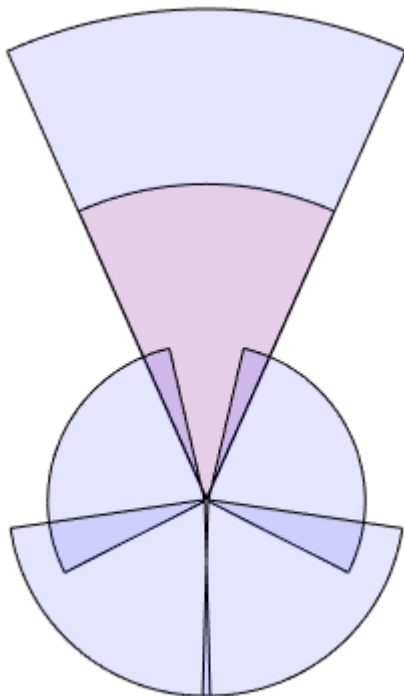
where ΔT is the time difference between steps k and $k + 1$, w_s and w_d are zero-mean Gaussian noise representing unknown acceleration in Frenet coordinates, and τ is a decaying constant.

This choice of coordinate in modeling the object motion allows you to integrate the highway reference path into the multi-object tracking framework. The integration of reference path acts as additional information for the tracker and allows the tracker to improve current state estimates as well as predicted trajectories of the estimated objects. You can obtain measurement model by first transforming the object state into Cartesian position and velocity and then converting them to respective measured quantities such as azimuth and range.

Setup

Scenario and Sensors

The scenario used in this example is created using the Driving Scenario Designer (Automated Driving Toolbox) and then exported to a MATLAB® function. The ego vehicle is mounted with 1 forward-looking radar and 5 cameras providing 360-degree coverage. The radar and cameras are simulated using the `drivingRadarDataGenerator` (Automated Driving Toolbox) and `visionDetectionGenerator` (Automated Driving Toolbox) System objects, respectively.



The entire scenario and sensor setup is defined in the helper function, `helperTrackingAndPlanningScenario`, attached with this example. You define the global plan describing the highway centerline using a `referencePathFrenet` object. As multiple algorithms in this example need access to the reference path, you define the `helperGetReferencePath` function, which uses a persistent object that can be accessed by any function.

```
rng(2022); % For reproducible results

% Setup scenario and sensors
[scenario, egoVehicle, sensors] = helperTrackingAndPlanningScenario();
```

Joint Probabilistic Data Association Tracker

You set up a joint probabilistic data association tracker using the `trackerJPDA` (Sensor Fusion and Tracking Toolbox) System object. You set the `FilterInitializationFcn` property of the tracker to `helperInitRefPathFilter` function. This helper function defines an extended Kalman filter, `trackerJPDA` (Sensor Fusion and Tracking Toolbox), used to estimate the state of a single object. Local functions inside the `helperInitRefPathFilter` file define the state transition as well as measurement model for the filter. Further, to predict the tracks at a future time for the motion planner, you use the `predictTracksToTime` (Sensor Fusion and Tracking Toolbox) function of the tracker.

```
tracker = trackerJPDA('FilterInitializationFcn',@helperInitRefPathFilter,...
    'AssignmentThreshold',[200 inf],...
    'ConfirmationThreshold',[8 10],...
    'DeletionThreshold',[5 5]);
```

Motion Planner

You use a similar highway trajectory motion planner as outlined in the “Highway Trajectory Planning Using Frenet Reference Path” on page 1-397 example. The motion planner uses a planning horizon of 5 seconds and considers three modes for sampling trajectories for the ego vehicle — cruise control, lead vehicle follow, and basic lane change. The entire process for generating an optimal trajectory is wrapped in the helper function, `helperPlanHighwayTrajectory`.

The helper function accepts an `dynamicCapsuleList` object as an input to find non-colliding trajectories. The collision checking is performed in the entire planning horizon at an interval of 0.5 seconds. As the track states vary with time, you update the `dynamicCapsuleList` object in the simulation loop using the `helperUpdateCapsuleList` function, attached with this example.

```
% Collision check time stamps
tHorizon = 5; % seconds
deltaT = 0.5; % seconds
tSteps = deltaT:deltaT:tHorizon;

% Create the dynamicCapsuleList object
capList = dynamicCapsuleList;
capList.MaxNumSteps = numel(tSteps) + 1;

% Specify the ego vehicle geometry
carLen = 4.7;
carWidth = 1.8;
rearAxleRatio = 0.25;
egoID = 1;
[egoID, egoGeom] = egoGeometry(capList,egoID);
```

```
% Inflate to allow uncertainty and safety gaps
egoGeom.Geometry.Length = 2*carLen; % in meters
egoGeom.Geometry.Radius = carWidth/2; % in meters
egoGeom.Geometry.FixedTransform(1,end) = -2*carLen*rearAxleRatio; % in meters
updateEgoGeometry(capList, egoID, egoGeom);
```

Run Simulation

In this section, you advance the simulation, generate sensor data and perform dynamic replanning using estimations about the surroundings. The entire process is divided into 5 main steps:

- 1 You collect simulated sensor data from radar and camera sensors.
- 2 You feed the sensor data to the JPDA tracker to estimate current state of objects.
- 3 You predict the state of objects using the `predictTracksToTime` function.
- 4 You update the object list for the planner and plan a highway trajectory.
- 5 You move the simulated ego vehicle on the planned trajectory.

```
% Create display for visualizing results
display = HelperTrackingAndPlanningDisplay;
```

```
% Initial state of the ego vehicle
refPath = helperGetReferencePath;
egoState = frenet2global(refPath, [0 0 0 0.5*3.6 0 0]);
helperMoveEgoToState(egoVehicle, egoState);
```

```
while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Step 1. Collect data
    detections = helperGenerateDetections(sensors, egoVehicle, time);

    % Step 2. Feed detections to tracker
    tracks = tracker(detections, time);

    % Step 3. Predict tracks in planning horizon
    timesteps = time + tSteps;
    predictedTracks = repmat(tracks, [1 numel(timesteps)+1]);
    for i = 1:numel(timesteps)
        predictedTracks(:,i+1) = predictTracksToTime(tracker, 'confirmed', timesteps(i));
    end

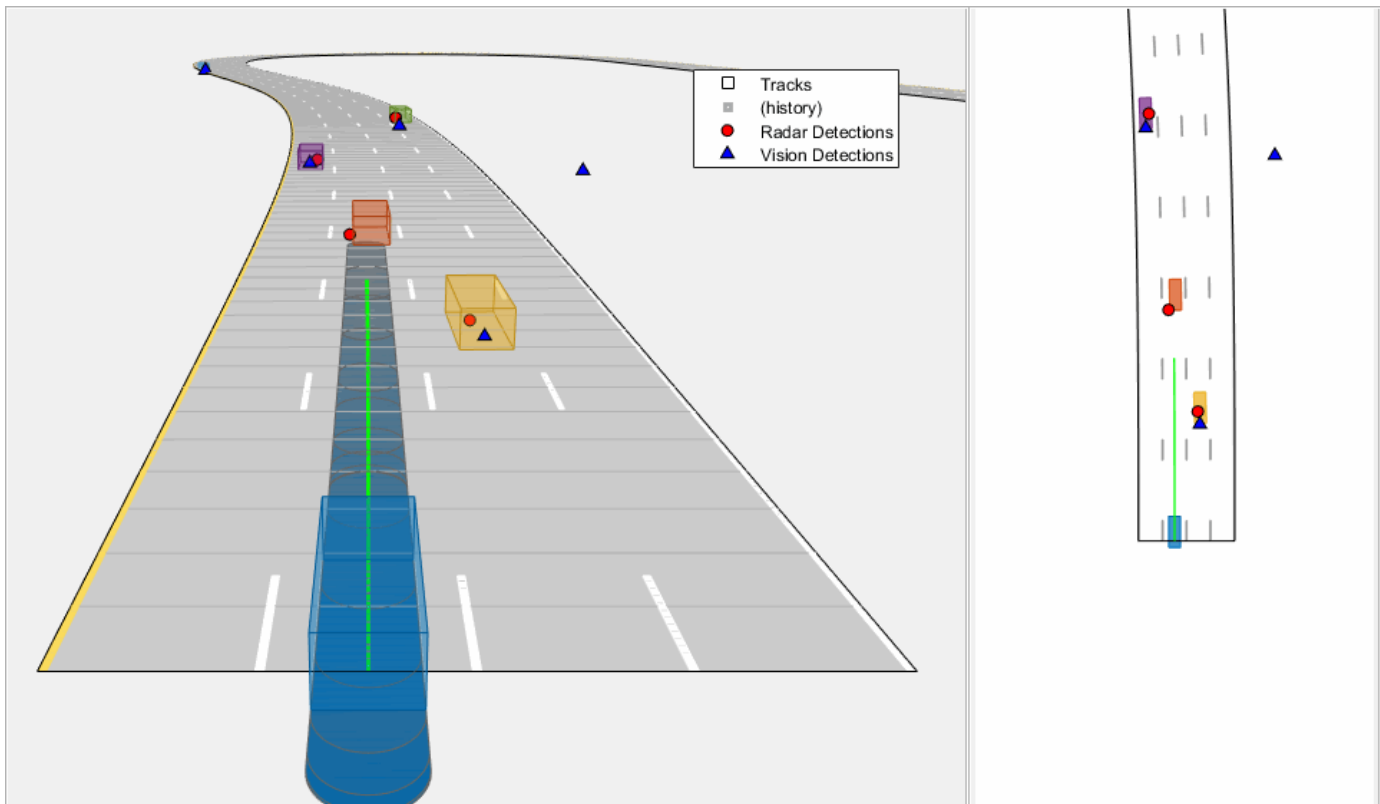
    % Step 4. Update capsule list and plan highway trajectory
    currActorState = helperUpdateCapsuleList(capList, predictedTracks);
    [optimalTrajectory, trajectoryList] = helperPlanHighwayTrajectory(capList, currActorState, egoVehicle);

    % Visualize the results
    display(scenario, egoVehicle, sensors, detections, tracks, capList, trajectoryList);

    % Step 5. Move ego on planned trajectory
    egoState = optimalTrajectory(2,:);
    helperMoveEgoToState(egoVehicle, egoState);
end
```

Results

In the animation below, you can observe the planned ego vehicle trajectories highlighted in green color. The animation also shows all other sampled trajectories for the ego vehicle. For these other trajectories, the colliding trajectories are shown in red, unevaluated trajectories are shown in grey, and kinematically-infeasible trajectories are shown in cyan color. Each track is annotated by an ID representing its unique identity. Notice that the ego vehicle successfully maneuvers around obstacles in the scene.

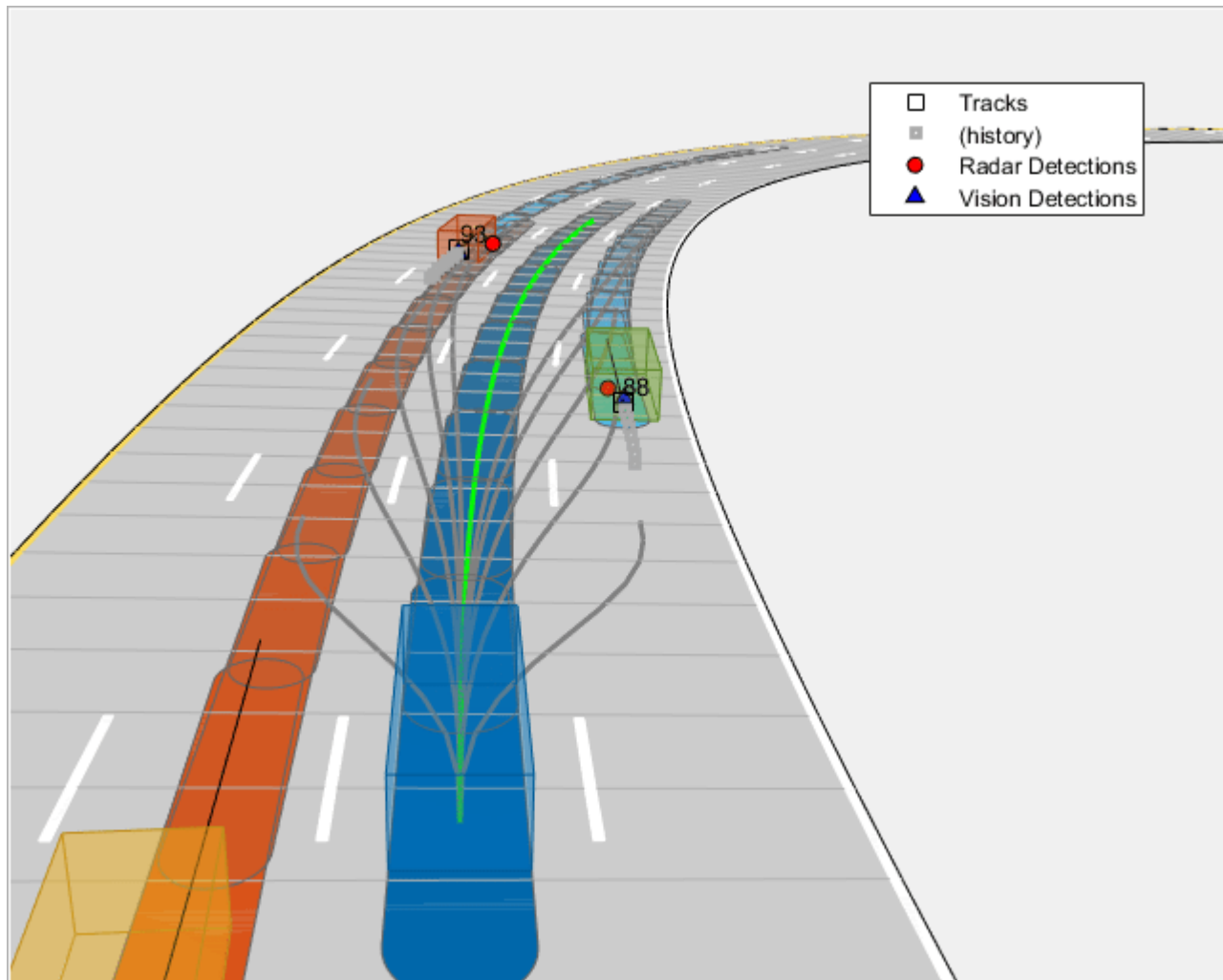


In the following sub-sections, you analyze the estimates from the tracker at certain time steps and understand how it impacts the choices made by the motion planner.

Road-integrated motion prediction

In this section, you learn how the road-integrated motion model allows the tracker to obtain more accurate long-term predictions about the objects on the highway. Shown below is a snapshot from the simulation taken at time = 30 seconds. Notice the trajectory predicted for the green vehicle to the right of the blue ego vehicle. The predicted trajectory follows the lane of the vehicle because the road network information is integrated with the tracker. If instead, you use a constant-velocity model assumption for objects, the predicted trajectory will follow the direction of instantaneous velocity and will be falsely treated as a collision by the motion planner. In this case, the motion planner can possibly generate an unsafe maneuver.

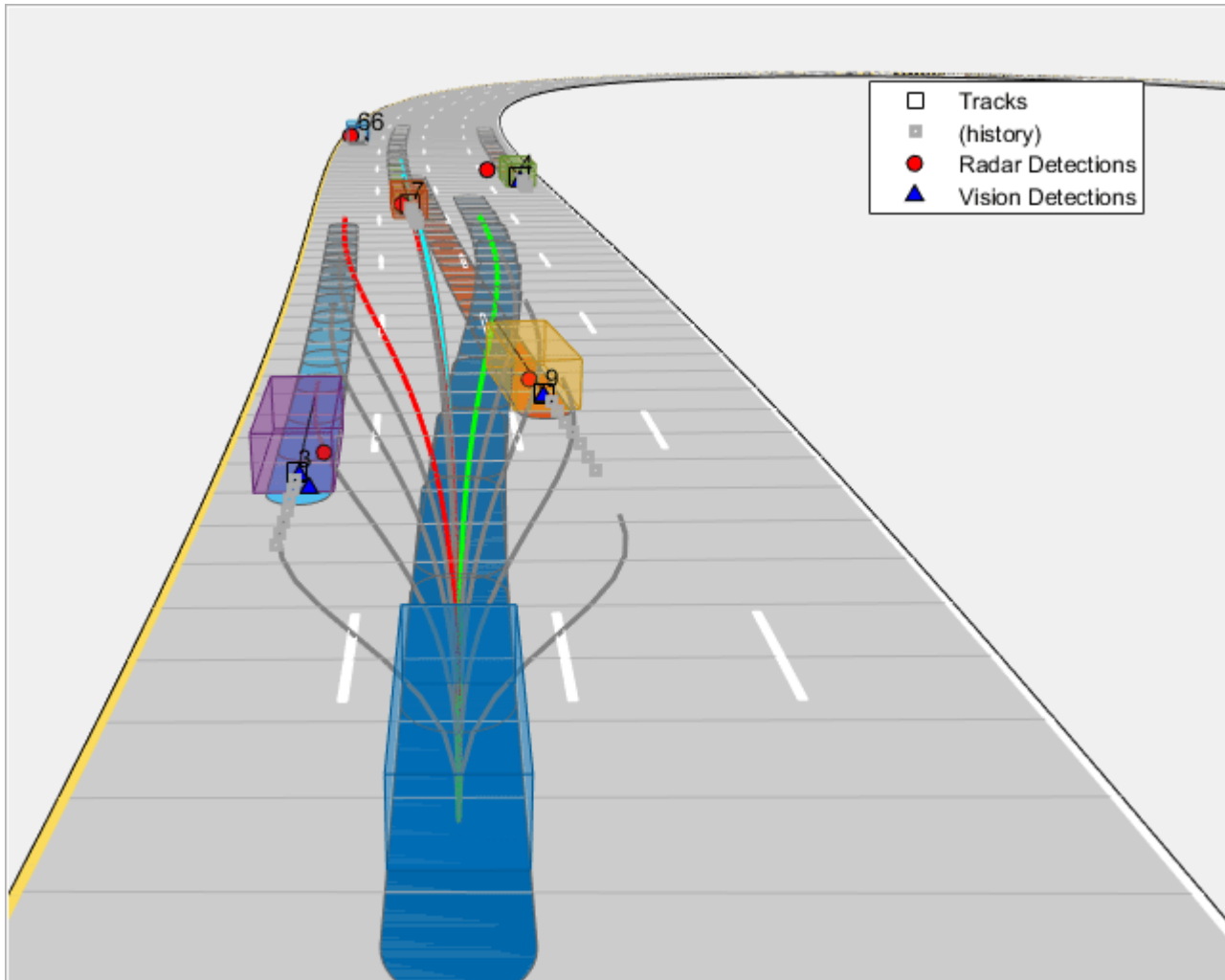
```
showSnaps(display,2,4); % Shows snapshot while publishing
```



Lane change prediction

In the first section, you learned how the lane change maneuvers are captured by using a decaying lateral velocity model of the objects. Now, notice the snapshot taken at time = 17.5 seconds. At this time, the yellow vehicle on the right side of the ego vehicle initiates a lane change and intends to enter the lane of the ego vehicle. Notice that its predicted trajectory captures this maneuver, and the tracker predicts it to be in the same lane as the ego vehicle at the end of planning horizon. This prediction informs the motion planner about a possible collision with this vehicle, thus the planner first proceeds to test feasibility for the ego vehicle to change lane to the left. However, the presence of purple vehicle on the left and its predicted trajectory causes the ego vehicle to make a right lane change. You can also observe these colliding trajectories colored as red in the snapshot below.

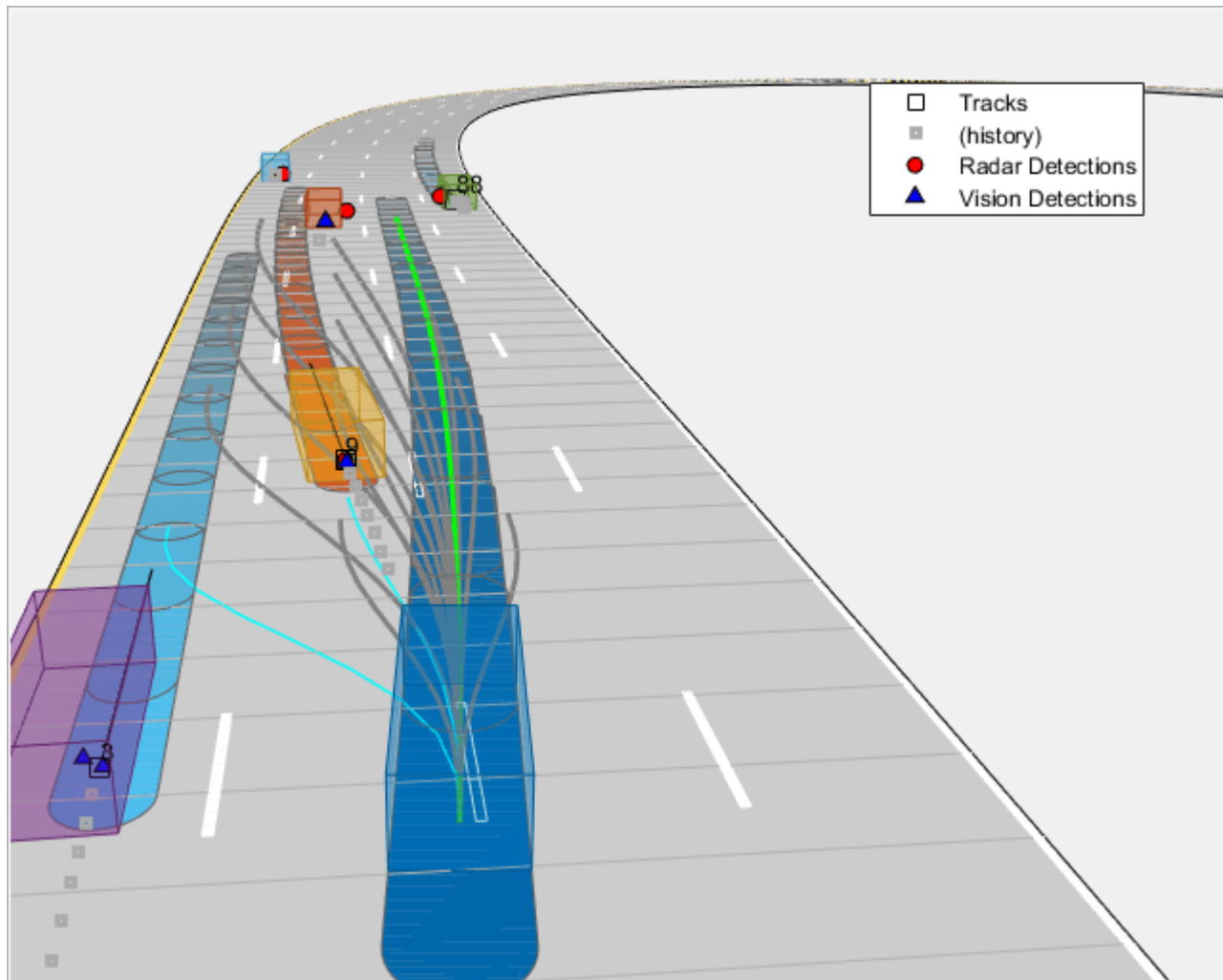
```
showSnaps(display,2,1); % Shows snapshot while publishing
```



Tracker imperfections

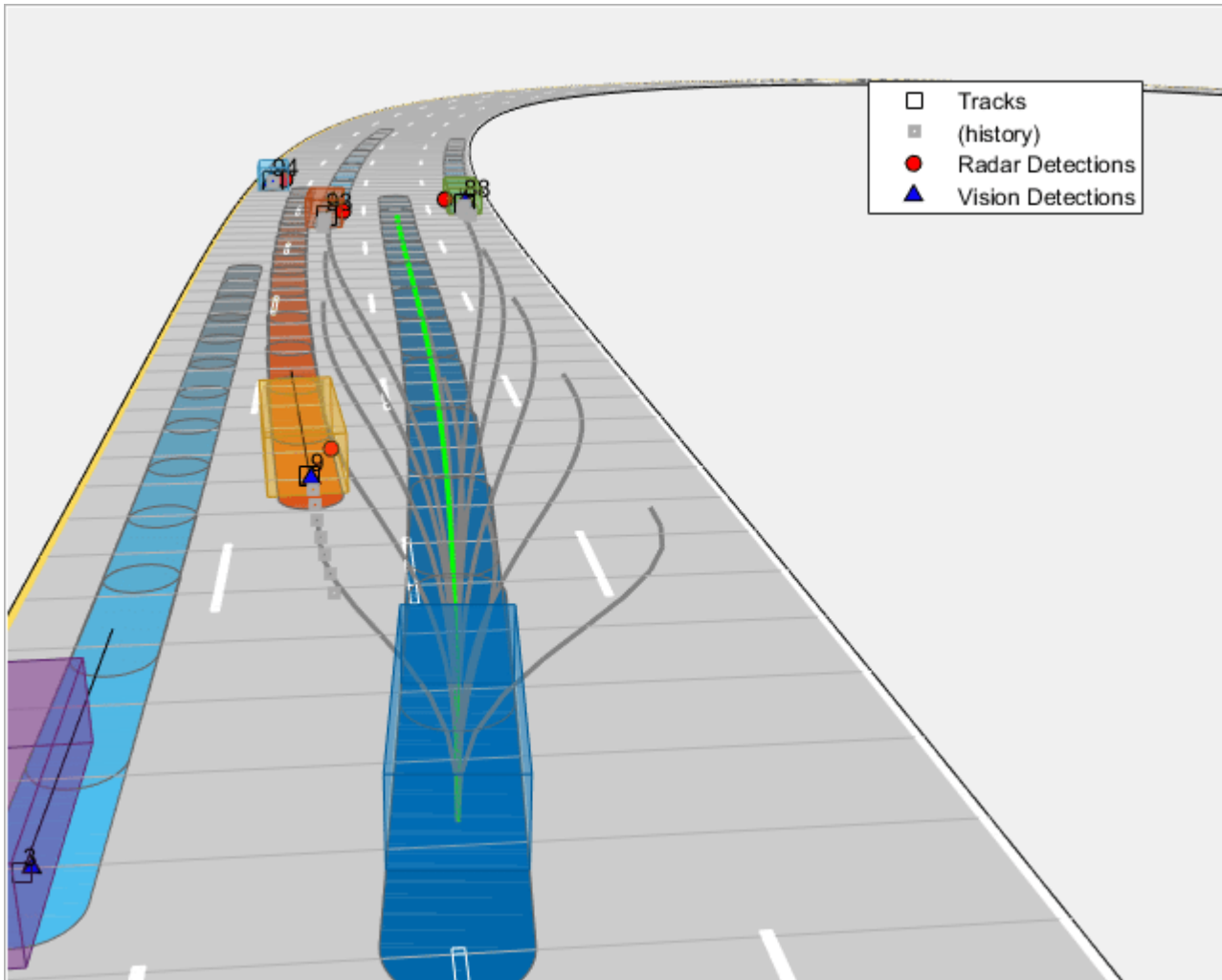
A multi-object tracker may have certain imperfections that can affect motion planning decisions. Specifically, a multi-object tracker can miss objects, report false tracks, or sometimes report redundant tracks. In the snapshot below taken at time = 20 seconds, the tracker drops tracks on two vehicles in front of the ego vehicle due to occlusion. In this particular situation, these missed targets are less likely to influence the decision of the motion planner due to their distance from the ego vehicle.

```
showSnaps(display,2,2); % Shows snapshot while publishing
```



However, as the ego vehicle approaches these vehicles, their influence on the ego vehicle's decision increases. Notice that the tracker is able to establish a track on these vehicles by time = 20.4 seconds, as shown in the snapshot below, thus making the system slightly robust to these imperfections. While configuring a tracking algorithm for motion planning, it is important to consider these imperfections from the tracker and tune the track confirmation and track deletion logics.

```
showSnaps(display,2,3); % Show snapshot while publishing
```



Summary

You learned how to use a joint probabilistic data association tracker to track vehicles using a Frenet reference path with radar and camera sensors. You configured the tracker to use highway map data to provide long term predictions about objects. You also used these long-term predictions to drive a motion planner for planning trajectories on the highway.

Supporting Functions

```
function detections = helperGenerateDetections(sensors, egoVehicle, time)
    detections = cell(0,1);
    for i = 1:numel(sensors)
        thisDetections = sensors{i}(targetPoses(egoVehicle),time);
        detections = [detections;thisDetections]; %#ok<AGROW>
    end

    detections = helperAddEgoVehicleLocalization(detections,egoVehicle);
    detections = helperPreprocessDetections(detections);
end

function detectionsOut = helperAddEgoVehicleLocalization(detectionsIn, egoPose)
```

```
defaultParams = struct('Frame','Rectangular',...
    'OriginPosition',zeros(3,1),...
    'OriginVelocity',zeros(3,1),...
    'Orientation',eye(3),...
    'HasAzimuth',false,...
    'HasElevation',false,...
    'HasRange',false,...
    'HasVelocity',false);

fNames = fieldnames(defaultParams);

detectionsOut = cell(numel(detectionsIn),1);

for i = 1:numel(detectionsIn)
    thisDet = detectionsIn{i};
    if iscell(thisDet.MeasurementParameters)
        measParams = thisDet.MeasurementParameters{1};
    else
        measParams = thisDet.MeasurementParameters(1);
    end

    newParams = struct;
    for k = 1:numel(fNames)
        if isfield(measParams,fNames{k})
            newParams.(fNames{k}) = measParams.(fNames{k});
        else
            newParams.(fNames{k}) = defaultParams.(fNames{k});
        end
    end

    % Add parameters for ego vehicle
    thisDet.MeasurementParameters = [newParams;newParams];
    thisDet.MeasurementParameters(2).Frame = 'Rectangular';
    thisDet.MeasurementParameters(2).OriginPosition = egoPose.Position(:);
    thisDet.MeasurementParameters(2).OriginVelocity = egoPose.Velocity(:);
    thisDet.MeasurementParameters(2).Orientation = rotmat(quaternion([egoPose.Yaw egoPose.Pitch e

    % No information from object class and attributes
    thisDet.ObjectClassID = 0;
    thisDet.ObjectAttributes = struct;
    detectionsOut{i} = thisDet;
end

end

function detections = helperPreprocessDetections(detections)
    % This function pre-process the detections from radars and cameras to
    % fit the modeling assumptions used by the tracker

    % 1. It removes velocity information from camera detections. This is
    % because those are filtered estimates and the assumptions from camera
    % may not align with defined prior information for tracker.
    %
    % 2. It fixes the bias for camera sensors that arise due to camera
    % projections for cars just left or right to the ego vehicle.
    %
```



```

% 3. It inflates the measurement noise for range-rate reported by the
% radars to match the range-rate resolution of the sensor
for i = 1:numel(detections)
    if detections{i}.SensorIndex > 1 % Camera
        % Remove velocity
        detections{i}.Measurement = detections{i}.Measurement(1:3);
        detections{i}.MeasurementNoise = blkdiag(detections{i}.MeasurementNoise(1:2,1:2),25)
        detections{i}.MeasurementParameters(1).HasVelocity = false;

        % Fix bias
        pos = detections{i}.Measurement(1:2);
        if abs(pos(1)) < 5 && abs(pos(2)) < 5
            [az, ~, r] = cart2sph(pos(1),pos(2),0);
            [pos(1),pos(2)] = sph2cart(az, 0, r + 0.7); % Increase range
            detections{i}.Measurement(1:2) = pos;
            detections{i}.MeasurementNoise(2,2) = 0.25;
        end
    else % Radars
        detections{i}.MeasurementNoise(3,3) = 0.5^2/4;
    end
end
end

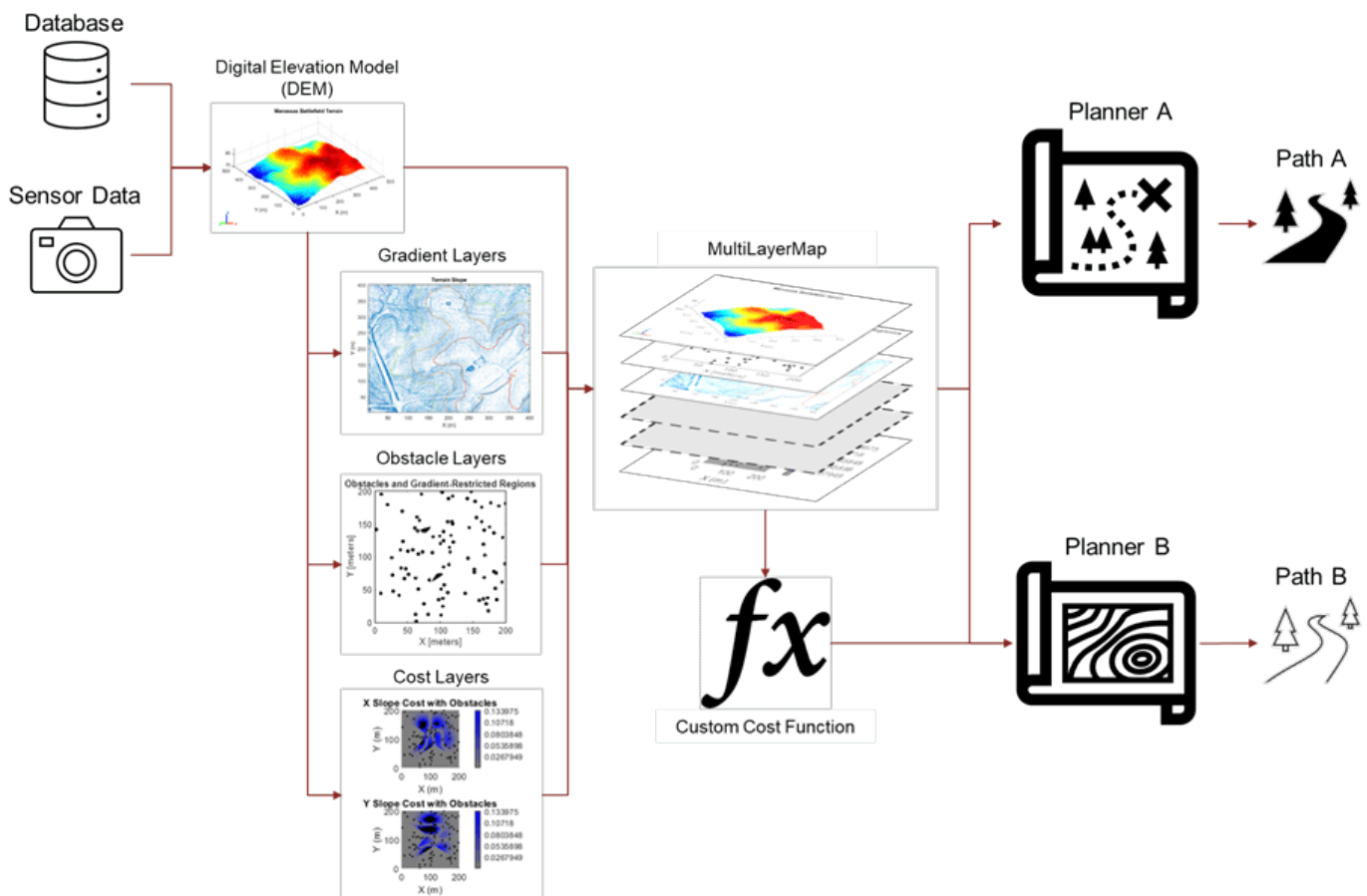
function helperMoveEgoToState(egoVehicle, egoState)
egoVehicle.Position(1:2) = egoState(1:2);
egoVehicle.Velocity(1:2) = [cos(egoState(3)) sin(egoState(3))]*egoState(5);
egoVehicle.Yaw = egoState(3)*180/pi;
egoVehicle.AngularVelocity(3) = 180/pi*egoState(4)*egoState(5);
end

```

Offroad Planning with Digital Elevation Models

This example shows how to process and store 2.5-D information, and presents various techniques for using it for an offroad path planner.

Mobile robot planning often involves finding the shortest path for a wheeled robot in the presence of obstacles. Planners often assume that the planning space is a 2-D Cartesian plane, with certain regions marked as off limits due to the presence of obstacles. When it comes to offroad vehicles, environments can also contain changes in elevation, turning this into a three-dimensional problem. Planning in higher-dimension spaces coincides with longer planning times, so an effective compromise can be to plan in a 2.5-D space using Digital Elevation Models (DEMs). This example shows the process of creating and testing planning heuristics in simulation, and then shows how to apply those heuristics to achieve 2.5-D path planning for an offroad planner with real-world data.



Define Planning Space

Create Planning Surface

Most environment models are created from sensors mounted on the robot, or retrieved from a database based on the estimated pose of the platform. Create a surface using peaks to act as the environment model to test the effect of different cost functions for the planner.

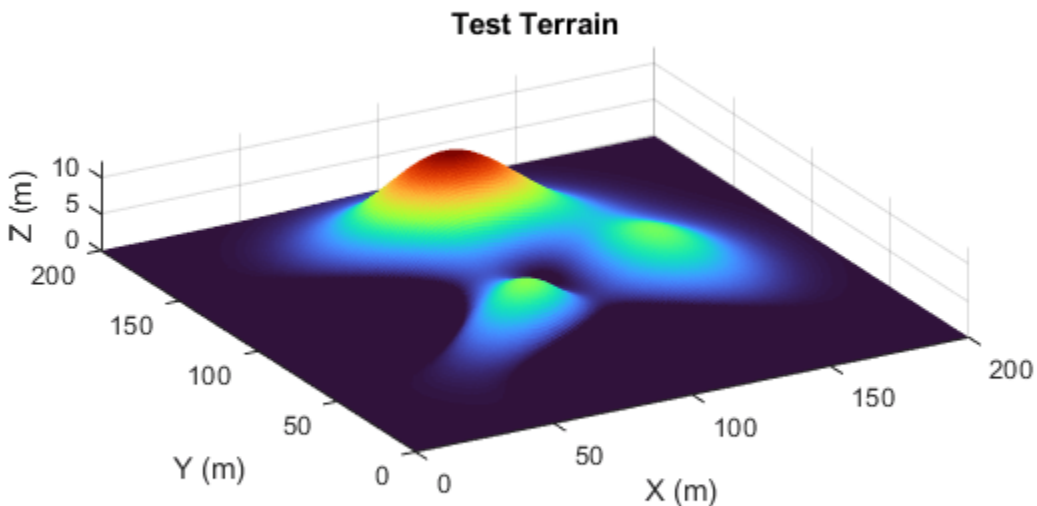
```
% Add helper folders to path
addpath("Analysis", "Data", "Heuristics", "MapProcessing");
```

```

% Create surface
[X,Y] = meshgrid(-100:100);
Z = peaks(max(size(X))*1.5); % Units in meters
X = X + 100; % Move X origin from center to corner
Y = Y + 100; % Move X origin from center to corner
Z(Z<0) = 0; % Discard valleys

% Visualize surface
figure
ax = gca;
surf(ax,X,Y,Z,EdgeColor="none")
title("Test Terrain")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
colormap("turbo")
ax.PlotBoxAspectRatio = [1 1 0.15];
view([-29.59 21.30])

```



Define Heuristics for 2.5-D Space

Next, define the heuristics. These heuristics utilize elevation data in different ways to create a cost, C , which may result in improvement versus path planning in only two dimensions:

exampleHelperZHeuristic

This cost function augments the planning state (XY) with the surface height (Z) before calculating distance. This should result in the shortest possible route through the 2.5-D manifold, balancing changes in elevation against distance to find the most direct path between two points:

$$C_Z = ||s^*_2 - s^*_1||, \text{ where } s^*_i = [x_i \ y_i \ z_i * w],$$

exampleHelperGradientHeuristic

This cost function adds a cost that scales with the steepness of the slope in the *same* direction as the heading. Unlike the Z heuristic, this penalty is not scaled by distance, and should therefore bias the planner towards routes that minimize immediate changes in elevation, potentially saving energy:

$$C_{\text{Slope}} = \begin{cases} ||s_2 - s_1|| + w * \left(\overline{|s_2 - s_1|}, |g_x \ g_y| \right) & \text{if slope} < \text{slope}_{\text{max}} \\ e^{\left(\overline{|s_2 - s_1|}, |g_x \ g_y| \right)} & \text{else} \end{cases}$$

exampleHelperRolloverHeuristic

This cost function is similar to `exampleHelperGradientHeuristic`, but by flipping the X and Y values of the spatial gradient, it penalizes motions whose direction is *perpendicular* to the slope. In other words, this function should minimize the risk of rollover by seeking routes with low roll angles:

$$C_{\text{Roll}} = \begin{cases} ||s_2 - s_1|| + w * \left(\overline{|s_2 - s_1|}, |g_y \ g_x| \right) & \text{if slope} < \text{slope}_{\text{max}} \\ e^{\left(\overline{|s_2 - s_1|}, |g_y \ g_x| \right)} & \text{else} \end{cases}$$

Each cost function includes a weighting variable that scales the accompanying heuristic. You can adjust the weight to prioritize the quickest 2-D route (weight = 0) or a potentially more energy-efficient path (weight = 4).

Discretize and Store Environment Information in Map Layers

With the cost functions defined, we know what information to retrieve from the environment. Discretely sample the 3-D surface and store the results in `mapLayer` objects. These provide transforms between Cartesian coordinates and functions for querying or modifying data, similar to the `binaryOccupancyMap` and `occupancyMap` objects.

Construct Z-Layer

You can take the height layer directly from the surface and store it in a `mapLayer` object, effectively serving as your Digital Elevation Model. Create a map layer to contain the height of the surface with ground clutter and obstacles removed.

```
% Query and store the Z-height
zLayer = mapLayer(flipud(Z), LayerName="Z");
```

Calculate Gradients and Convert to Cost

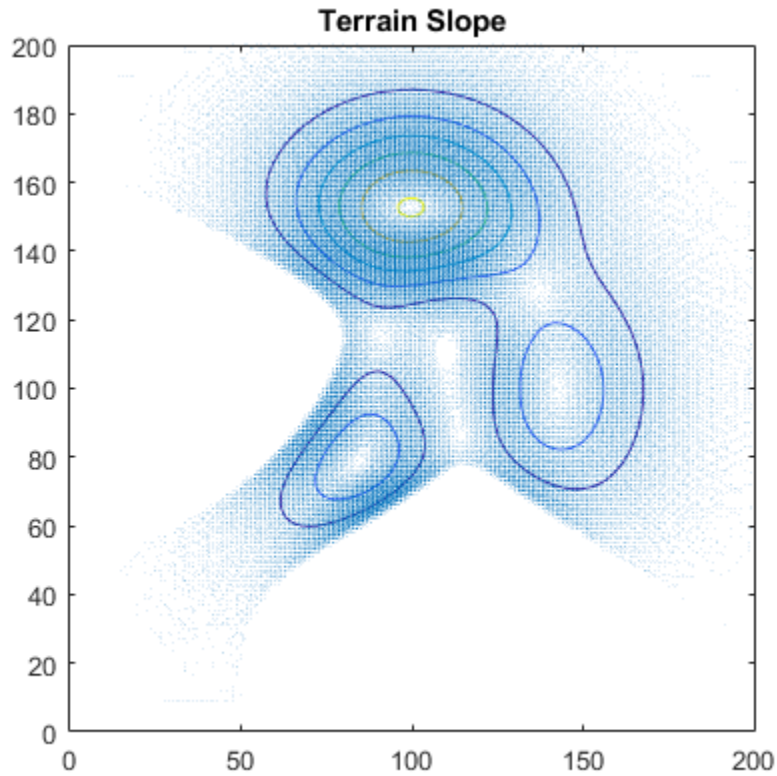
Calculate the gradient and cost information from the Z-layer. First, calculate the X and Y gradients of the surface, and use the `contour` function to visualize the gradients as a contour map.

```
[gx,gy] = gradient(Z);
figure
contour(X,Y,Z)
title("Terrain Slope")
```

```

hold on
quiver(X,Y,gx,gy)
axis equal
hold off

```



Store the X and Y gradients in mapLayer objects.

```

dzdx = mapLayer(flipud(gx),LayerName="dzdx");
dzdy = mapLayer(flipud(gy),LayerName="dzdy");

```

Define the function `exampleHelperGradientToCost`, which maps gradient values to a cost. To make it more intuitive to apply a cost-weighting variable, use slopes to determine the cost weighting. Slope values from 0 to a maximum slope scale linearly in the range [0 1], and values larger than the maximum slope scale exponentially:

$$C = \begin{cases} \frac{|m|}{m_{\max}} & , 0 \leq |m| \leq m_{\max} \\ e^{\left(\frac{|m|}{m_{\max}} - 1\right)} & , m_{\max} \leq |m| \end{cases}$$

where m is slope. Define the maximum incline angle as 15 degrees, and convert this to the maximum slope.

```

maxinclineangle = 15; % Degrees
maxSlope = tand(maxinclineangle); % Max preferred slope for vehicle

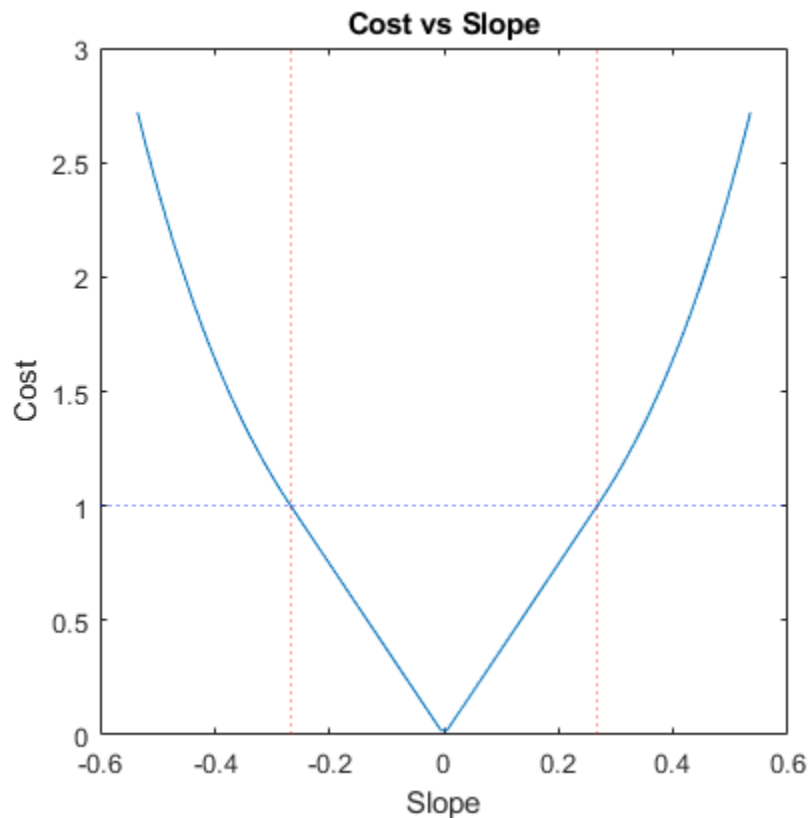
```

Visualize the unweighted slope-to-cost function.

```

slope2cost = @(x)exampleHelperGradientToCost(maxSlope,flipud(x));
slope = linspace(-2*maxSlope,2*maxSlope,100);
plot(slope,slope2cost(slope))
hold on
xline(maxSlope*[-1 1],"r:")
yline(1,"b:")
title("Cost vs Slope")
xlabel("Slope")
ylabel("Cost")
axis square

```



Calculate costs using the slope function, and store them in map layers.

```

xCost = mapLayer(slope2cost(gx),LayerName="xCost");
yCost = mapLayer(slope2cost(gy),LayerName="yCost");
diagCost = mapLayer(slope2cost(sqrt(gx.^2+gy.^2)),LayerName="diagCost");

```

Terrain Obstacles

In addition to elevation and slope information, you must represent areas that are offlimits, either due to high slope or obstacles. Use `binaryOccupancyMap` or `occupancyMap` objects to represent such regions for a `plannerAStarGrid` planner.

Generate a random matrix, then mark all elements with values above a specified threshold as obstacle locations. To represent the obstacle probabilities for testing, set the obstacle probabilities to random values.

```
rng(4) % Set RNG seed for repeatable results
obstacleprob = rand(max(size(X)));
```

Create a `binaryOccupancyMap` containing the obstacles and inflated terrain.

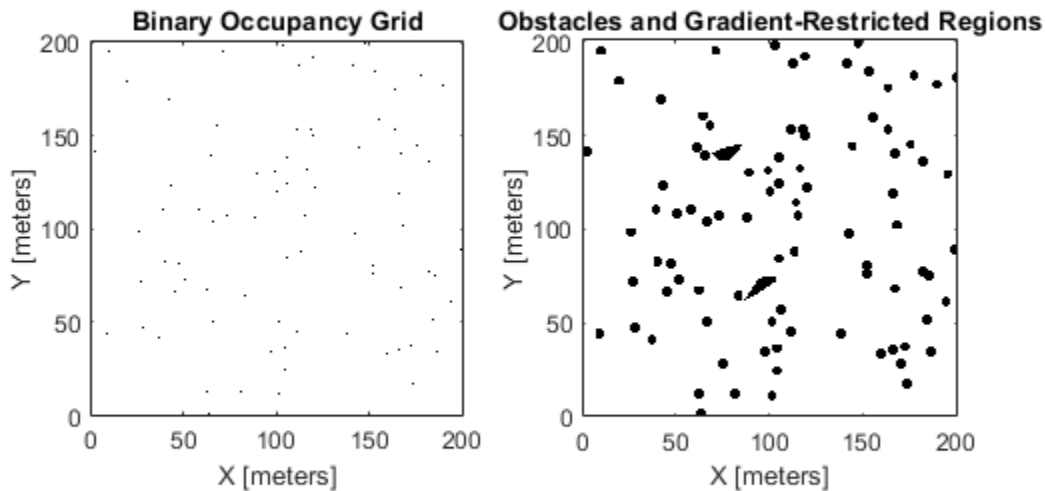
```
obstacleMap = binaryOccupancyMap(flipud(obstacleprob>0.998),LayerName="obstacles");
terrainObstacles = binaryOccupancyMap(obstacleMap,LayerName="terrainObstacles");
inflate(terrainObstacles,2);
```

Check each cell against the three slope constraints, and block off cells that violate all three slope constraints.

```
obstaclesAndTerrain = double(getMapData(terrainObstacles));
mInvalidSlope = getMapData(xCost) > 1 & getMapData(yCost) > 1 & getMapData(diagCost) > 1;
obstaclesAndTerrain(mInvalidSlope) = true;
setMapData(terrainObstacles,obstaclesAndTerrain);
```

Display a map containing the raw obstacles and a map containing inflated obstacles and invalid terrain.

```
figure
show(obstacleMap,Parent=subplot(1,2,1))
show(terrainObstacles,Parent=subplot(1,2,2))
title("Obstacles and Gradient-Restricted Regions")
```



Combine Individual Map Layers

To organize your data and keep your layers synchronized, add your layers to a `multiLayerMap`. This ensures that updating shared properties of one layer also updates the others, so that the data always represents the same region of Cartesian space.

```
costMap = multiLayerMap({zLayer dzdx dzdy terrainObstacles obstacleMap xCost yCost diagCost});
```

Make maps egocentric by updating the `GridOriginInLocal` value of the X cost map, and then restore the map to its original settings.

```
xCost.GridOriginInLocal = -xCost.GridSize/xCost.Resolution/2
```

```
xCost =
  mapLayer with properties:

  mapLayer Properties
      DataSize: [201 201]
      DataType: 'double'
      DefaultValue: 0
      GridSize: [201 201]
      LayerName: 'xCost'
  GridLocationInWorld: [-100.5000 -100.5000]
  GridOriginInLocal: [-100.5000 -100.5000]
  LocalOriginInWorld: [0 0]
      Resolution: 1
      XLocalLimits: [-100.5000 100.5000]
      YLocalLimits: [-100.5000 100.5000]
      XWorldLimits: [-100.5000 100.5000]
      YWorldLimits: [-100.5000 100.5000]
  GetTransformFcn: []
  SetTransformFcn: []
```

```
% Restore map to original settings
costMap.GridOriginInLocal = [0 0];
```

Visualize obstacles on top of the three slope-based cost layers. In these graphs, note how the slope threshold affects the cost:

- Gray — Free space
- Shades of blue — Varying costs below the soft threshold (the closer to gray, the lower the cost)
- Shades of red — Varying costs above the soft threshold (brighter red, higher cost)
- Black — Obstacles and cells that violate the slope threshold in all directions

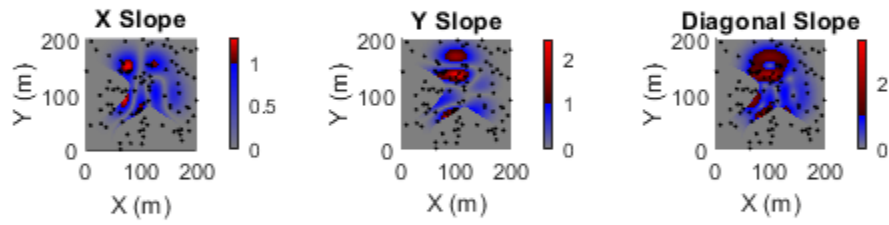
```
figure(5)
obstaclesAndTerrain(obstaclesAndTerrain < 1) = nan; % Turn 0s to NaNs for plotting on costmaps

show(visualizationHelper,xCost,subplot(1,3,1),hold="on",title="X Slope",Threshold=1);
surface(flipud(obstaclesAndTerrain),FaceColor=[0 0 0])

show(visualizationHelper,yCost,subplot(1,3,2),hold="on",title="Y Slope",Threshold=1);
surface(flipud(obstaclesAndTerrain),FaceColor=[0 0 0])

show(visualizationHelper,diagCost,subplot(1,3,3),hold="on",title="Diagonal Slope",Threshold=1);
surface(flipud(obstaclesAndTerrain),FaceColor=[0 0 0])
sgtitle("Unweighted Costs with Obstacles")
```


Unweighted Costs with Obstacles



Plan Paths Using 2.5-D Heuristics

Create Planners

Create a set of planners, each using one of the defined heuristics, and compare their planning results.

Define the start and goal locations and convert to them into grid coordinates.

```
startposition = [54 131];
goalposition = [143 104];

start = world2grid(costMap,startposition);
goal = world2grid(costMap,goalposition);
```

Assign a G and H weight, and create the planner for each custom heuristic.

```
gWeight = 1;
hWeight = 1;

defaultPlanner = plannerAStarGrid(getLayer(costMap,"terrain0bstacles"));

heightAwarePlanner = plannerAStarGrid(getLayer(costMap,"terrain0bstacles"), ...
    GCostFcn=@(s1,s2)exampleHelperZHeuristic(costMap,gWeight,s1,s2), ...
    HCostFcn=@(s1,s2)exampleHelperZHeuristic(costMap,hWeight,s1,s2));

gradientAwarePlanner = plannerAStarGrid(getLayer(costMap,"terrain0bstacles"), ...
    GCostFcn=@(s1,s2)exampleHelperGradientHeuristic(costMap,gWeight,s1,s2), ...
    HCostFcn=@(s1,s2)exampleHelperGradientHeuristic(costMap,hWeight,s1,s2));
```

```

rolloverAwarePlanner = plannerAStarGrid(getLayer(costMap,"terrainObstacles"), ...
    GCostFcn=@(s1,s2)exampleHelperRolloverHeuristic(costMap,gWeight,s1,s2), ...
    HCostFcn=@(s1,s2)exampleHelperRolloverHeuristic(costMap,hWeight,s1,s2));

```

Plan Paths

Plan the paths with each heuristic from the start to the goal and store them in a cell array. Use `tic` and `toc` to measure the time it takes for each heuristic planner to plan their path.

```

% Planners return a sequence of IJ cells
paths = cell(4,1);
pathNames = ["Default","Elevation-Aware Path","Gradient-Aware Path","Rollover-Aware Path"];
planningTime = nan(4,1);
tic;
paths{1} = plan(defaultPlanner,start,goal);
planningTime(1) = toc;
paths{2} = plan(heightAwarePlanner,start,goal);
planningTime(2) = toc-planningTime(1);
paths{3} = plan(gradientAwarePlanner,start,goal);
planningTime(3) = toc-planningTime(2);
paths{4} = plan(rolloverAwarePlanner,start,goal);
planningTime(4) = toc-planningTime(3);

```

Compare and Analyze Results

Compare the performance of each planner by defining a set of metrics and visualizing the motion along each path. This example models the platform as a simple ground vehicle, and makes some simplifying assumptions related to the energy required to operate the robot. Edit these parameters to see how the planning results are affected.

Define Robot and Scenario Parameters

```

% Define simulation parameters
scenarioParams.Gravity = 9.81; % m/s^2
scenarioParams.UpdateRate = 1/60; % Hz

% Define vehicle parameters for estimating power consumption
scenarioParams.Robot.Mass = 70; % kg, no payload
scenarioParams.Robot.Velocity = 0.6; % m/s
scenarioParams.Robot.AmpHour = 24; % A-hr
scenarioParams.Robot.Vnom = 24; % V, Voltage
scenarioParams.Robot.RegenerativeBrakeEfficiency = 0;

```

Define Performance Metrics

For the analysis, use 3-D path distance, travel time, planning time, and energy expenditure as a rough measure of path efficiency. Also define the minimum, maximum, and average pitch and roll angles of the vehicle, to compare the relative quality of each path. For more information, see `exampleHelperCalculateMetrics`.

```

% Display result
figure
animateResults = true;
statTable = exampleHelperCalculateMetrics(costMap,scenarioParams,paths,planningTime,pathNames)

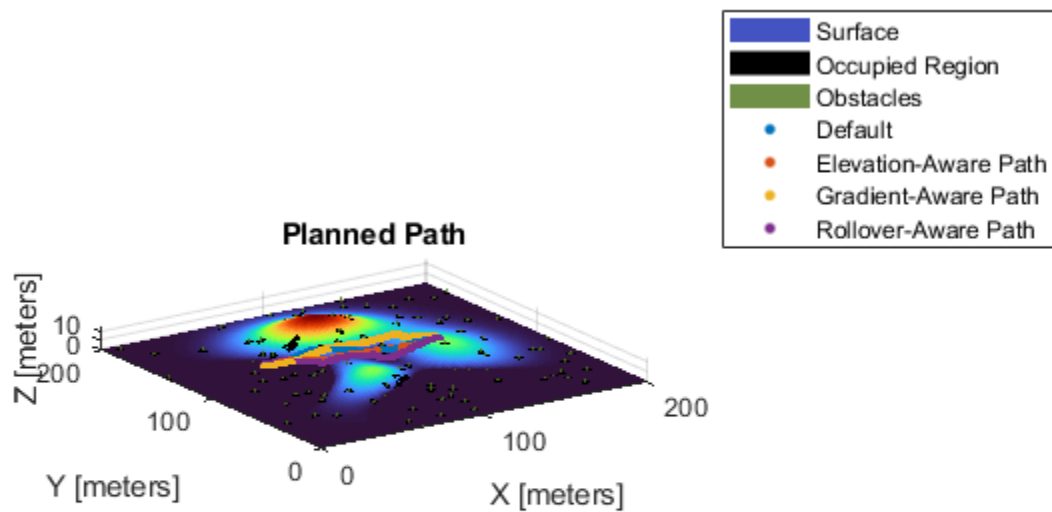
```

```
statTable=4x8 table
```

	Distance (m)	Power (W)	Travel Time (s)	Planning Time (s)
Default	101.2	226.85	2.8112	0.060131
Elevation-Aware Path	100.7	218.13	2.7972	2.7577
Gradient-Aware Path	109.97	214.76	3.0547	18.582
Rollover-Aware Path	103.19	218.03	2.8665	11.886

Visualize the path results using `exampleHelperVisualizeResults`. This helper function projects all of the previously generated paths onto the 3-D surface.

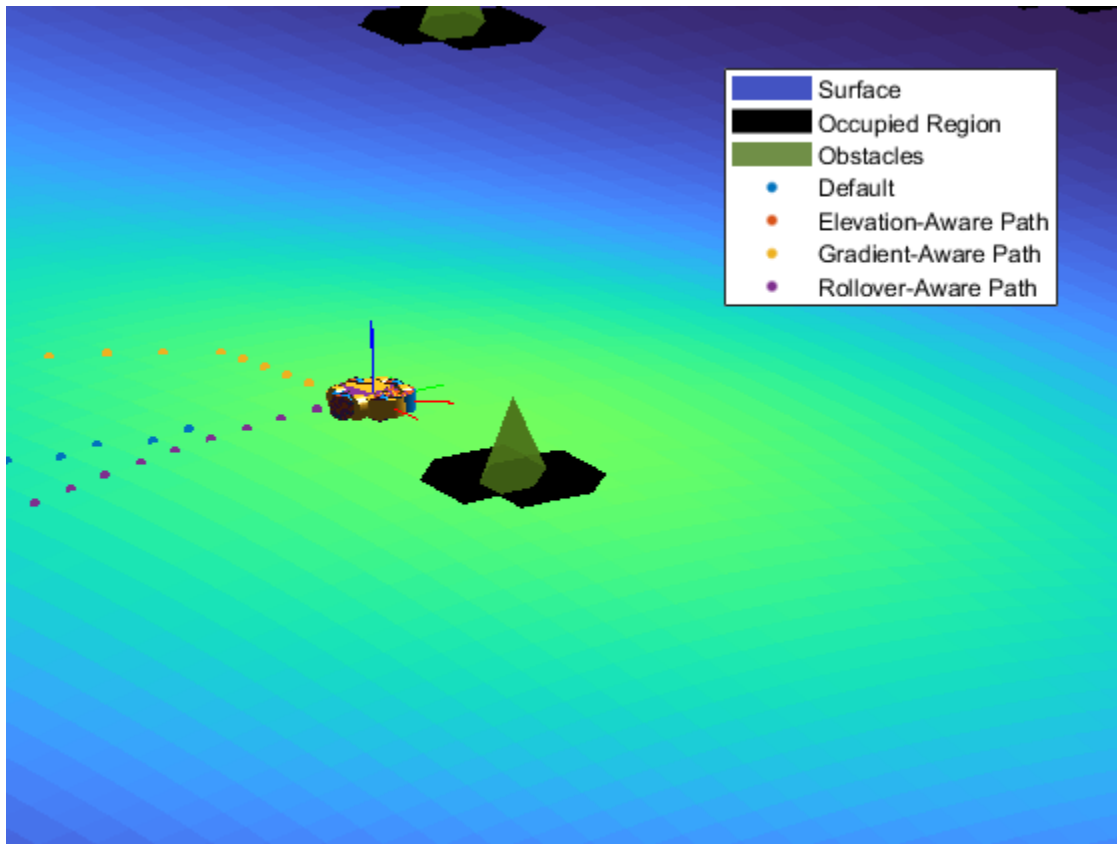
```
exampleHelperVisualizeResults(gca,statTable,scenarioParams,costMap,paths,pathNames,~animateResults)
```



Visualize Path Results

Visualize the robot following each of the planned paths.

```
figure
exampleHelperVisualizeResults(gca,statTable,scenarioParams,costMap,paths,pathNames,animateResults)
```



Analyzing the results, the elevation-aware planner offers the best result for the lowest 3-D distance and time, and keeps power consumption relatively low, but the elevation-aware planner does little to steer the robot away from regions with steep inclines or high roll angles. The default planner also takes a short route, even with height included in the distance metric, but it expends additional energy ascending and descending slopes that the elevation-aware planner avoids, resulting in a higher power consumption.

The gradient-based planners trade distance and time for safer routes that consume less power compared to the other planners. The gradient-aware planner minimizes the time spent climbing and descending hills, whereas the rollover-aware planner seeks a path that keeps it level while traversing terrain. These results are reflected in the pitch and roll metrics, where the gradient-aware planner shows the lowest average pitch, and the rollover-aware planner shows maximum and average roll angles well below the other planners.

Plan on Real-World Digital Elevation Model Data

Now that you have tested the heuristics on an artificial surface, test them using real-world elevation data.

Import Real-World Data

You can import Digital Elevation Data from a variety of sources. This example uses a TIF file downloaded from USGS National Land Cover Database that contains elevation data for Manassas National Battlefield Park, Virginia in the US. If you want to download this file yourself, follow these steps:

- 1 In the database address bar, search "Manassas National Battlefield Park, VA".
- 2 Zoom to the region of interest.
- 3 On the **Datasets** tab, under **Data**, select **Elevation Products (3-DEP) > 1 meter DEM**, and click **Search Products**.
- 4 Download the file covering the region for which you want elevation data.

Once downloaded, you can extract the elevation data from the TIF file using the `readgeoraster` function of Mapping Toolbox™:

Note: This code requires Mapping Toolbox

```
% Load the terrain data
```

```
[elevation,info] = readgeoraster("USGS_1m_x28y430_VA_Fairfax_County_2018.tif",OutputType="double",res = 2.5;
```

```
% Select a portion of the terrain data
```

```
Zinit = flipud(elevation(700:1100,800:1200));  
[Xinit,Yinit] = meshgrid(1:length(Zinit));
```

```
% Interpolate for better matching with image resolution
```

```
[Xreal,Yreal] = meshgrid(1:(1/res):length(Zinit));  
Zreal = interp2(Xinit,Yinit,Zinit,Xreal,Yreal);
```

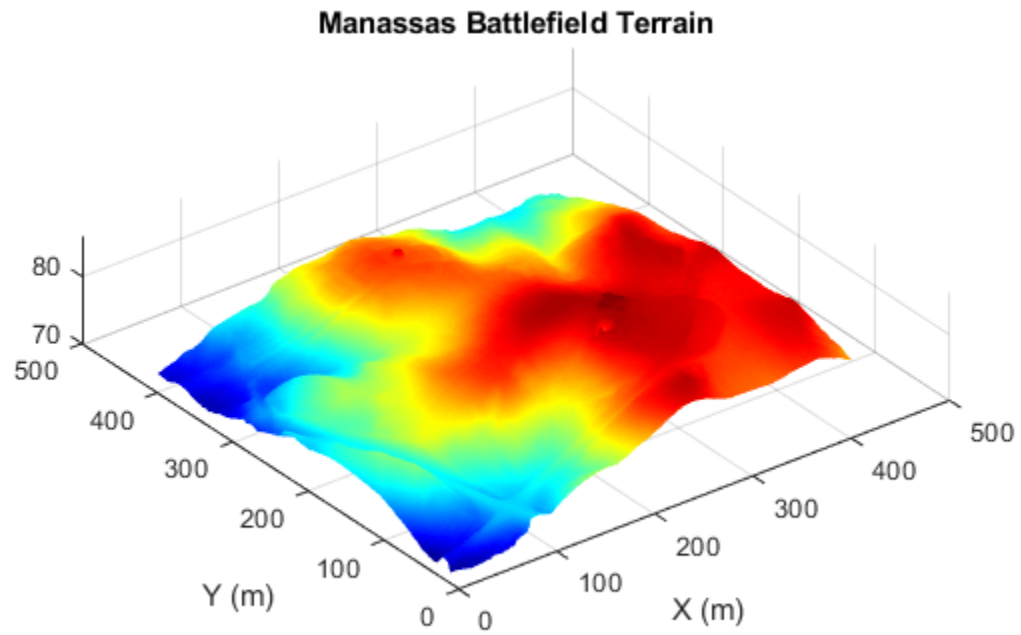
For this example, the data has been processed for you and stored in a MAT file. Load the post-processed data for the subregion of interest.

```
% Load saved surface data and resolution
```

```
load("manassasData.mat");
```

```
% Visualize surface
```

```
surf(Xreal,Yreal,Zreal,EdgeColor="none")  
formatAxes(visualizationHelper,title="Manassas Battlefield Terrain",colormap="jet")  
pbaspect([1 1 0.2])
```



Load satellite imagery of the same location

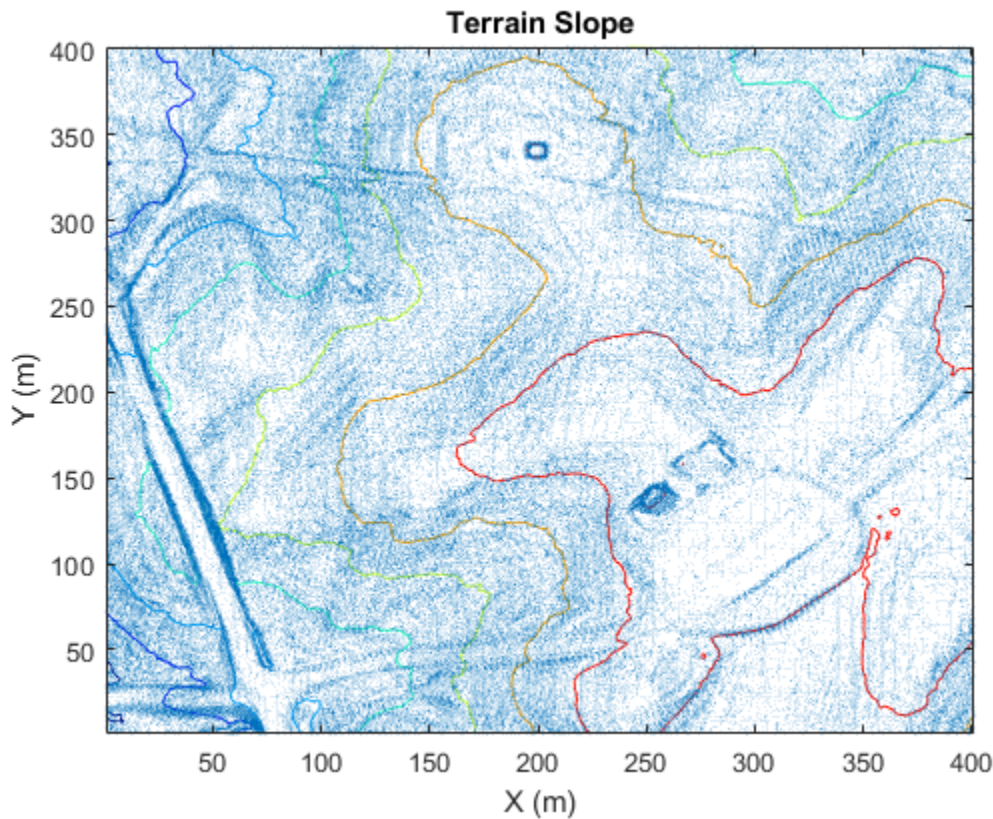
```
img = imread("visitorcenter_satellitesq.jpg");  
imgResized = imresize(img,[1001 1001]);  
imshow(img)
```



Create Data Layers

Repeat the same process as in the Calculate Gradients and Convert to Cost on page 1-0 section to create the gradient and gradient cost layers.

```
figure
[gxReal,gyReal] = gradient(Zreal);
contour(Xreal,Yreal,Zreal)
formatAxes(visualizationHelper,title="Terrain Slope",hold="on",colormap="jet")
quiver(Xreal,Yreal,gxReal,gyReal)
```



Convert the satellite image into a terrain-obstacle map.

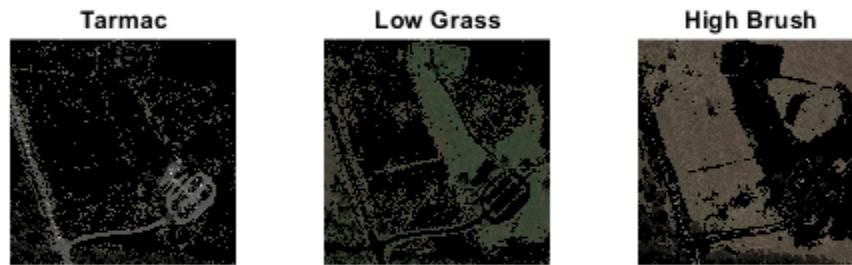
```
% Construct data layers
```

```
zLayerReal = mapLayer(flipud(Zreal),Resolution=res,LayerName="Z");
dzdxReal = mapLayer(flipud(gxReal),Resolution=res,LayerName="dzdx");
dzdyReal = mapLayer(flipud(gyReal),Resolution=res,LayerName="dzdy");
xCostReal = mapLayer(slope2cost(flipud(gxReal)),Resolution=res,LayerName="xCost");
yCostReal = mapLayer(slope2cost(flipud(gyReal)),Resolution=res,LayerName="yCost");
diagCostReal = mapLayer(slope2cost(flipud(sqrt(gxReal.^2+gyReal.^2))),Resolution=res,LayerName="diagCost");
```

Because an overhead satellite image is available, you can attempt to apply cost or invalidate areas of the map based on the absence or presence of vegetation. Use three filters: one to highlight colors with low spectral intensity (roads), another that focuses on greens (open fields), and a third that highlights browns and yellows (tall grass).

```
% Use thresholding to segment the image
```

```
[BWroads,roadRGB] = exampleHelperHighlightRoad(imgResized);
[BWfields,fieldRGB] = exampleHelperHighlightField(imgResized);
[BWbrush,brushRGB] = exampleHelperHighlightBrush(imgResized);
figure
imshow(roadRGB,Parent=subplot(1,3,1))
title("Tarmac")
imshow(fieldRGB,Parent=subplot(1,3,2))
title("Low Grass")
imshow(brushRGB,Parent=subplot(1,3,3))
title("High Brush")
```

Combine these three maps to create a terrain effort cost with the following weights for each type of terrain:

- Road or Tarmac — 0
- Low Grass — 1
- High Brush — 3

Count the number of these categories for which each cell qualifies, and calculate the cost for each cell.


```
count = double(BWroads) + double(BWfields) + double(BWbrush);
terrainCost = (double(BWfields) + double(BWbrush)*3)./count;
```

Replace NaN values with values interpolated from neighbors.

```
[ii,jj] = meshgrid(1:size(terrainCost,1),1:size(terrainCost,2));
inan = isnan(terrainCost);
terrainCost(inan) = max(griddata(ii(~inan),jj(~inan),terrainCost(~inan),ii(inan),jj(inan),"linear",0.5)));
```

For simplicity, this example marks any terrain with a slope or surface cost over a given threshold as occupied, but these costs could be incorporated into existing or new cost functions as well.

Insert the terrain costs into a layer and define a maximum terrain cost. Changing the maximum terrain cost can affect what terrain the binary occupancy map sees as an obstacle.

```
terrainLayer = mapLayer(terrainCost,Resolution=res,LayerName="surfaceCost");
maxTerrainCost = 2.5  ;
```

Convert the terrain costs into a binaryOccupancyMap object.

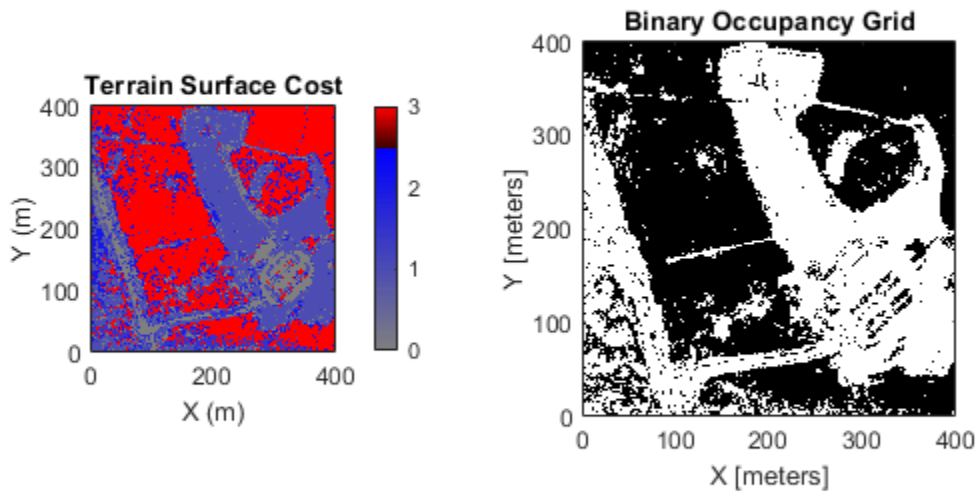
```
terrainObstaclesReal = binaryOccupancyMap(terrainLayer.getMapData >= maxTerrainCost,Resolution=Resolution);
```

Inflate slope-blocked cells and set them as obstacles.

```
inflate(terrainObstaclesReal, .25);
terrainData = getMapData(terrainObstaclesReal);
mInvalidSlope = getMapData(xCostReal) > 1 & getMapData(yCostReal) > 1 & getMapData(diagCostReal) > 1;
terrainData(mInvalidSlope) = true;
setMapData(terrainObstaclesReal, terrainData);
```

Display the terrain costs and occupancy map side-by-side.

```
figure
show(visualizationHelper, terrainLayer, subplot(1,2,1), title="Terrain Surface Cost", Threshold=maxTerrainCost);
show(terrainObstaclesReal, Parent=subplot(1,2,2))
```



Combine the cost map layers into a multiLayerMap object.

```
costMapReal = multiLayerMap({zLayerReal dzdxReal dzdyReal terrainObstaclesReal xCostReal yCostReal});
```

Create a planning configuration by configuring the g-cost and h-cost functions. Explore how different configurations impact the planned path.

Configure G-Cost Function

The function set to the `GCostFcn` property of `plannerAStarGrid` calculates the distance traveled (or cost) between two nodes in the planning graph. The planner then adds this cost to the accumulated cost between the root node and the current node to form the g-cost:

$$g_{\text{cost}_i} = \sum_{n=0}^{i-1} g_{\text{node}_{n-1}, \text{node}_n} + g_{\text{CostFcn}}(\text{node}_{i-1}, \text{node}_i)$$

```
gCostType =  ;
gFcnChoices = exampleHelperFcnOptions("Function Type",gCostType);
gWeightRange = exampleHelperFcnOptions("Weight Range",gCostType,"g");
gWLow = gWeightRange(1);
gWHigh = gWeightRange(2);
gStep = gWeightRange(3);
gWeight =  ;
vArgs = exampleHelperFcnOptions("Cost Function Inputs",gCostType);
gCostSetting = exampleHelperFcnOptions("Function Selection", ,vArgs{:});
```

Configure H-Cost Function

The `hCostFcn` of the planner calculates the *cost-to-go*, or estimated cost, between the current node and the goal node. The h-cost serves as a heuristic to guide the planner during exploration. In order to satisfy the optimality guarantee of A*, it should also be an underestimate of the cost-to-go:

$$h_{\text{cost}_i} = h_{\text{CostFcn}}(\text{node}_i, \text{node}_{\text{goal}})$$

```
hCostType =  ;
hFcnChoices = exampleHelperFcnOptions("Function Type",hCostType);
hWeightRange = exampleHelperFcnOptions("Weight Range",hCostType);
hWLow = hWeightRange(1);
hWHigh = hWeightRange(2);
hStep = hWeightRange(3);
hWeight =  ;
vArgs = exampleHelperFcnOptions("Cost Function Inputs",hCostType);
hCostSetting = exampleHelperFcnOptions("Function Selection", ,vArgs{:});
```

Add the g-cost and h-cost together to calculate the f-cost. The A* planner uses the f-cost to prioritize the next node to explore in the graph:

$$f_{\text{cost}_i} = g_{\text{cost}_i} + h_{\text{cost}_i}$$

Construct the A* planner with the selected cost functions.

```
aStarPlanner = plannerAStarGrid(getLayer(costMapReal,"terrainObstacles"),hCostSetting{:},gCostSetting{:});
```

Plan Path

Define start and goal positions and convert them to grid coordinates. Then, plan a path between the start and goal positions.

```
startposition = [290 300];
goalposition = [95 165];
```

```
start = world2grid(costMapReal,startposition);
goal = world2grid(costMapReal,goalposition);

tic;
pathReal = plan(aStarPlanner,start,goal);
planningTimeReal = toc;
```

Review Results

Calculate and view the results of the planned path.

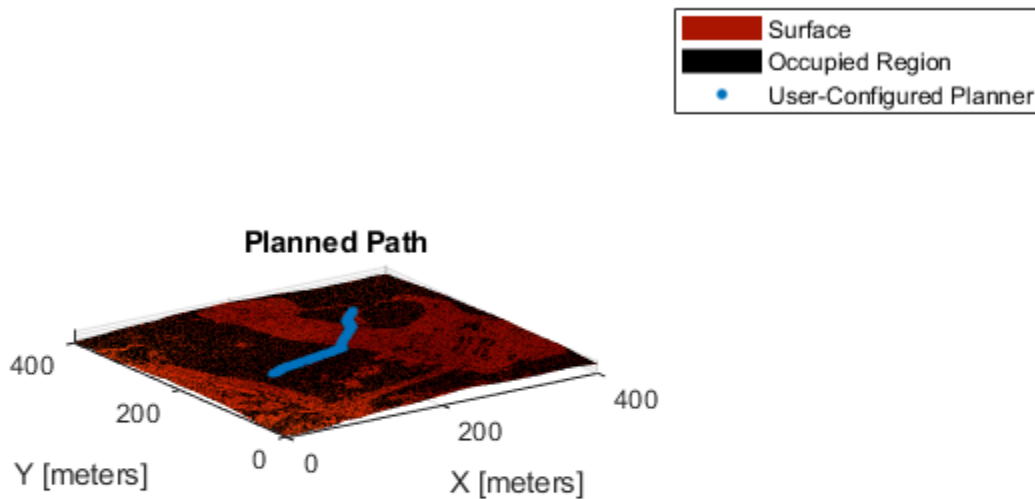
```
statTableReal = exampleHelperCalculateMetrics(costMapReal,scenarioParams,pathReal,planningTimeReal);
```

statTableReal=1x8 table

	Distance (m)	Power (W)	Travel Time (s)	Planning Time (s)
User-Configured Planner	264.16	197.87	7.3377	0.91902

Display the planned path on a figure of the area.

```
figure
exampleHelperVisualizeResults(gca,statTableReal,scenarioParams,costMapReal,{pathReal},"User-Conf
```



This example introduced the concept of offroad planning for mobile robots. You learned how to apply 2-D planners to 3-D searches by supplying custom cost functions that leverage 2.5-D information.

This 2.5-D information often comes in the form of Digital Elevation Models, which can be sourced from databases like the USGS National Land Cover Database, or constructed in real-time from onboard sensors. Lastly, this example showed you how this information can be stored in maps, how to construct different cost functions, and how to analyze the impact of those functions on path efficiency and quality.

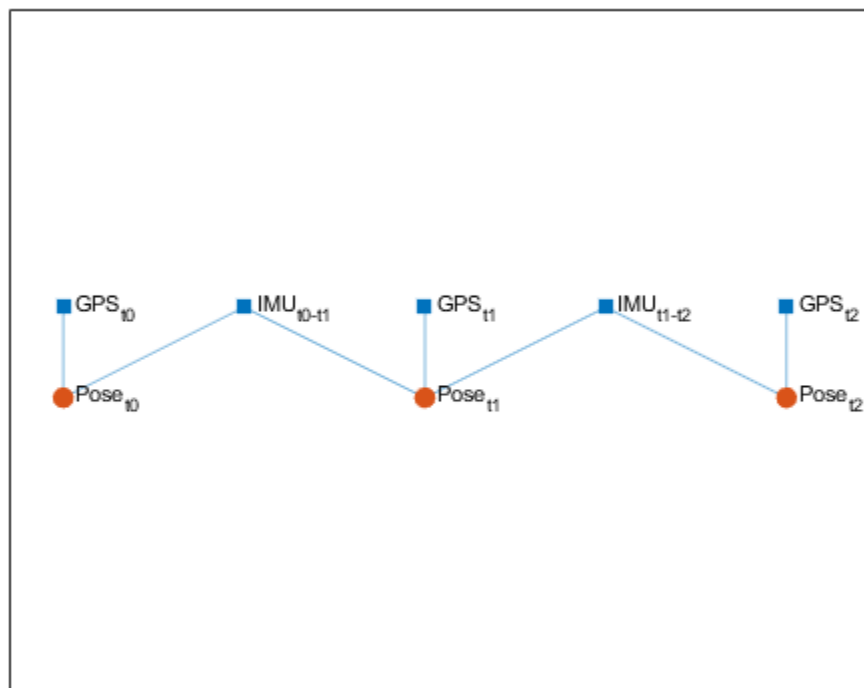
Factor Graph-Based Pedestrian Localization with IMU and GPS Sensors

This example shows how to estimate the position of a pedestrian using logged sensor data from an inertial measurement unit (IMU) and Global Positioning System (GPS) receiver and a factor graph.

A factor graph is a bipartite graph, or bigraph. A bipartite graph contains vertices that can be divided into two disjoint and independent sets. The first set of vertices are factors and contain fixed measurements, usually from a sensor, and a corresponding uncertainty matrix for each measurement. The second set of vertices are state nodes and update during factor graph optimization. The goal of optimization is to minimize every cost function defined for each connection of a factor and a state.

This example helper includes a function that plots a basic factor graph consisting of five factors and three state nodes, shown as blue squares and red circles respectively:

```
exampleHelperPlotBasicFactorGraph
```



The first factor, a GPS measurement at time t_0 , is connected to the first state node representing a pose at the same time t_0 . The second factor is all the IMU measurements between time t_0 and t_1 , and connects to both the first state node, a pose at t_0 , and the second state node, a pose at t_1 . During the optimization of the factor graph, the GPS and IMU factors constrain their connected pose states such that the new state value minimizes the value of the cost function corresponding to each factor.

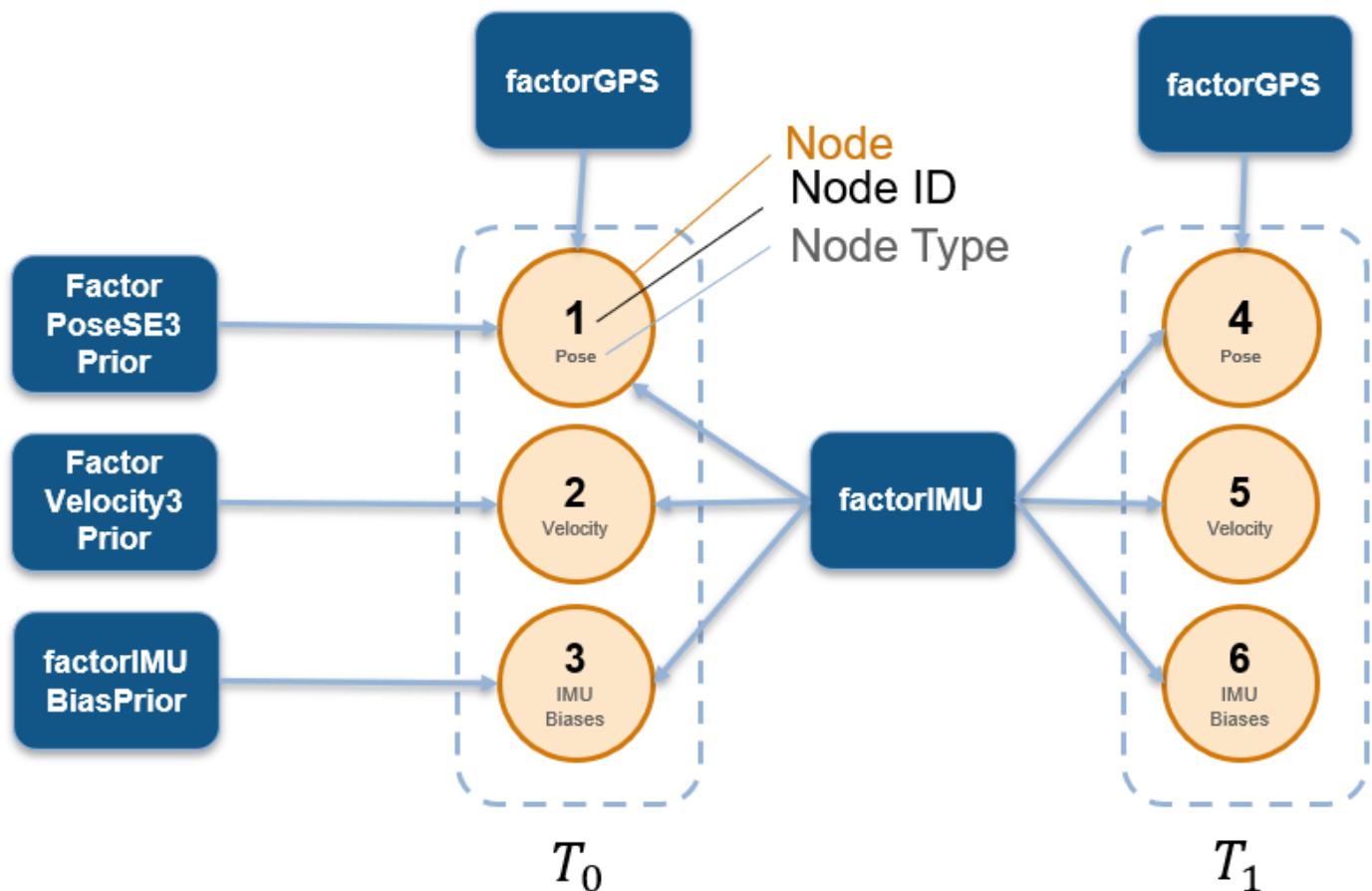
Load IMU and GPS Sensor Log File

Load a MAT file containing IMU and GPS sensor data, `pedestrianSensorDataIMUGPS`, and extract the sampling rate and noise values for the IMU, the sampling rate for the factor graph optimization, and the estimated position reported by the onboard filters of the sensors. This MAT file was created by logging data from a sensor held by a pedestrian walking on a pathway around the MathWorks campus.

```
load("pedestrianSensorDataIMUGPS.mat","sensorData","imuFs","imuNoise", ...
     "numGPSSamplesPerOptim","posLLA","initOrient","initPos","numIMUSamplesPerGPS");
```

Create Factor Graph

This diagram shows the structure of a factor graph for processing the sensor data. For each iteration of the loop in the Estimate Pedestrian Position on page 1-0 section, the time step updates and then advances to the next GPS reading. The process repeats until all of the sensor data has been processed. The diagram shows the relationship between the state nodes and factors for one pair of time steps or one iteration of the for loop.



Create a vector to keep track of the current node IDs for each pair of time steps.

```
nodesPerTimeStep = [exampleHelperFactorGraphNodes.Pose; ...
                    exampleHelperFactorGraphNodes.Velocity; ...
                    exampleHelperFactorGraphNodes.IMUBias];
numNodesPerTimeStep = numel(nodesPerTimeStep);
```

```
currNodeIDs = nodesPerTimeStep + [0 numNodesPerTimeStep];
currNodeIDs = currNodeIDs(:).';
```

Create time index variables to easily access the current node IDs vector. The initial node IDs for pose, velocity, and IMU biases are defined in the helper function `exampleHelperFactorGraphNodes`. This makes the initialization process of this example clearer.

```
t0IndexPose = exampleHelperFactorGraphNodes.Pose;
t1IndexPose = t0IndexPose + numNodesPerTimeStep;
t0IndexVel = exampleHelperFactorGraphNodes.Velocity;
t1IndexVel = t0IndexVel + numNodesPerTimeStep;
t0IndexBias = exampleHelperFactorGraphNodes.IMUBias;
t1IndexBias = t0IndexBias + numNodesPerTimeStep;
```

Create the factor graph as a `factorGraph` object and the three prior factors as a `factorPoseSE3Prior`, `factorVelocity3Prior`, and `factorIMUBiasPrior` object. Initialize the initial position, orientation, velocity, and IMU biases using the initial position and orientation from the onboard filter. The information matrix `Information` uses large values because they come from ground truth measurements. Add the prior factors to the factor graph using the `addFactor` function.

```
G = factorGraph;
prevPose = [initPos compact(initOrient)]; % x y z | qw qx qy qz
prevVel = [0 0 0]; % m/s
prevBias = [0 0 0 0 0 0]; % Gyroscope and accelerometer biases.

fPosePrior = factorPoseSE3Prior(currNodeIDs(t0IndexPose),Measurement=prevPose, ...
    Information=diag([4e4 4e4 4e4 1e4 1e4 1e4]));
fVelPrior = factorVelocity3Prior(currNodeIDs(t0IndexVel),Measurement=prevVel, ...
    Information=100*eye(3));
fBiasPrior = factorIMUBiasPrior(currNodeIDs(t0IndexBias),Measurement=prevBias, ...
    Information=1e6*eye(6));

addFactor(G, fPosePrior);
addFactor(G, fVelPrior);
addFactor(G, fBiasPrior);
```

Set the state nodes connected to the prior factors to the initial values.

```
nodeState(G, currNodeIDs(t0IndexPose), prevPose);
nodeState(G, currNodeIDs(t0IndexVel), prevVel);
nodeState(G, currNodeIDs(t0IndexBias), prevBias);
```

Get the local geodetic origin for the navigation frame from the initial GPS reading.

```
lla0 = sensorData{1}.GPSReading;
```

Create solver options as a `factorGraphSolverOptions` object, to use when optimizing the graph.

```
opts = factorGraphSolverOptions(MaxIterations=100);
```

Create a vector to store all the state node ID numbers that correspond to poses.

```
numGPSSamples = numel(sensorData);
poseIDs = zeros(1, numGPSSamples);
```

Estimate Pedestrian Position

Process the logged sensor data and extract the pedestrian position estimate from the optimized factor graph. Create a plot to visualize the estimated position.


```
visHelper = exampleHelperVisualizeFactorGraphAndFilteredPosition;
```

Perform these operations for each GPS sample in the sensor data log:

- 1 Store the current pose ID.
- 2 Create an IMU factor.
- 3 Create a GPS factor.
- 4 Add the factors to the factor graph.
- 5 Set initial node states using previous state estimates and a state prediction from the IMU measurements.
- 6 Optimize the factor graph.
- 7 Advance one time step by updating the previous state estimates to the current state and incrementing the current node IDs.

```
for ii = 1:numGPSSamples
% 1. Store the current pose ID.
poseIDs(ii) = currNodeIDs(t0IndexPose);

% 2. Create an IMU factor.
fIMU = factorIMU(currNodeIDs,imuFs, ...
    imuNoise.GyroscopeBiasNoise, ...
    imuNoise.AccelerometerBiasNoise, ...
    imuNoise.GyroscopeNoise, ...
    imuNoise.AccelerometerNoise, ...
    sensorData{ii}.GyroReadings,sensorData{ii}.AccelReadings);

% 3. Create a GPS factor.
fGPS = factorGPS(currNodeIDs(t0IndexPose),HDOP=sensorData{ii}.HDOP, ...
    VDOP=sensorData{ii}.VDoP, ...
    ReferenceLocation=lla0, ...
    Location=sensorData{ii}.GPSReading);

% 4. Add the factors to the graph.
addFactor(G,fIMU);
addFactor(G,fGPS);

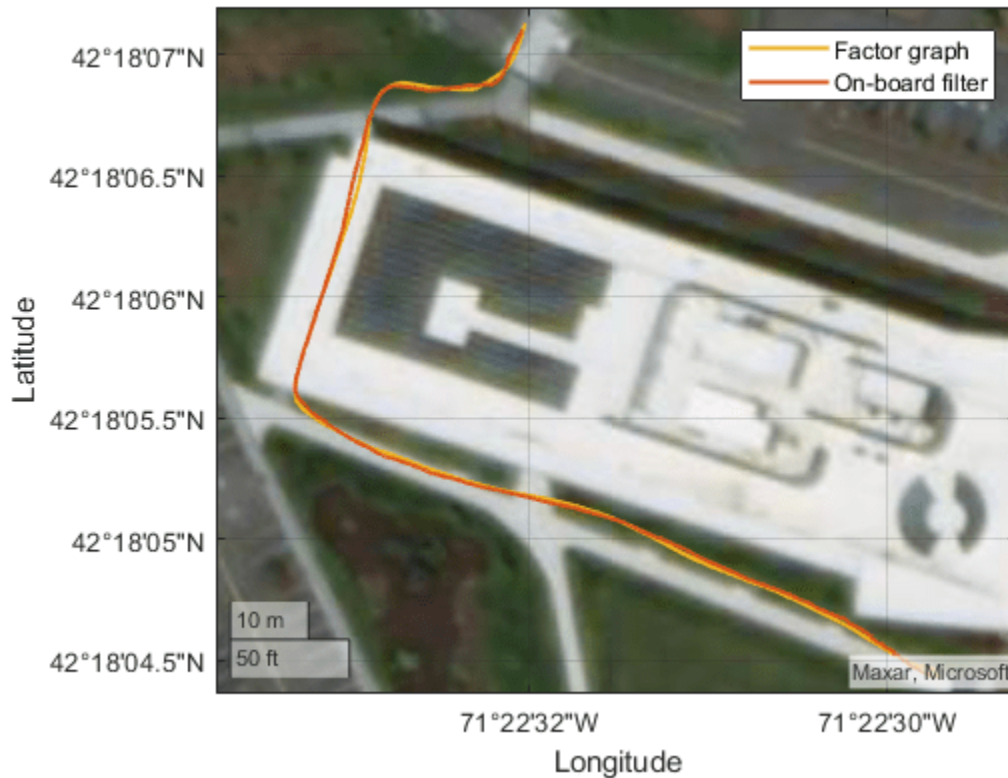
% 5. Set initial node states using previous state estimates and a state prediction from the
[predictedPose,predictedVel] = predict(fIMU,prevPose,prevVel,prevBias);
nodeState(G,currNodeIDs(t1IndexPose),predictedPose);
nodeState(G,currNodeIDs(t1IndexVel),predictedVel);
nodeState(G,currNodeIDs(t1IndexBias),prevBias);

% 6. Optimize the factor graph.
if (mod(ii,numGPSSamplesPerOptim) == 0) || (ii == numGPSSamples)
    solnInfo = optimize(G,opts);

    % Visualize the current position estimate.
    updatePlot(visHelper,ii,G,poseIDs,posLLA,lla0,numIMUSamplesPerGPS);
    drawnow
end

% 7. Advance one time step by updating the previous state estimates to the current ones and
prevPose = nodeState(G,currNodeIDs(t1IndexPose));
prevVel = nodeState(G,currNodeIDs(t1IndexVel));
prevBias = nodeState(G,currNodeIDs(t1IndexBias));
```

```
currNodeIDs = currNodeIDs + numNodesPerTimeStep;
end
```



The factor graph successfully estimates the position of the pedestrian compared to the position estimated by the onboard filter. To verify this, calculate the RMS error of the plotted 3-D position:

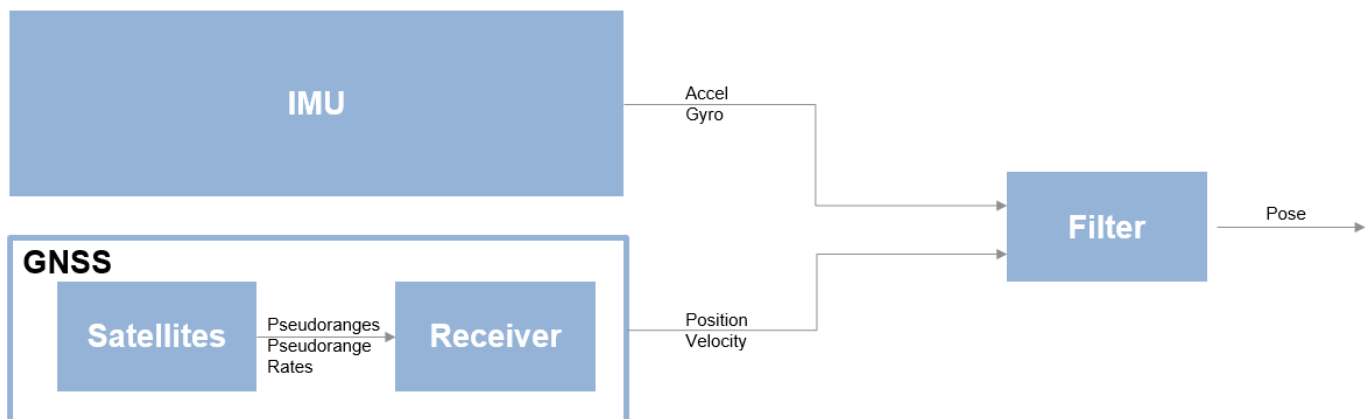
```
onBoardPos = lla2enu(posLLA,lla0,"ellipsoid");
onBoardPos = onBoardPos(cumsum(numIMUSamplesPerGPS-1),:); % Convert to sampling rate of factor graph
estPos = zeros(numel(poseIDs),3);
for ii = 1:numel(poseIDs)
    currPose = nodeState(G, poseIDs(ii));
    estPos(ii,:) = currPose(1:3);
end
posDiff = estPos - onBoardPos;
posDiff(isnan(posDiff(:,1)),:) = []; % Remove any missing position readings from the onboard filter
posRMS = sqrt(mean(posDiff.^2));
fprintf("3D Position RMS: X: %.2f, Y: %.2f, Z: %.2f (m)\n",posRMS(1),posRMS(2),posRMS(3));

3D Position RMS: X: 1.02, Y: 0.71, Z: 3.64 (m)
```

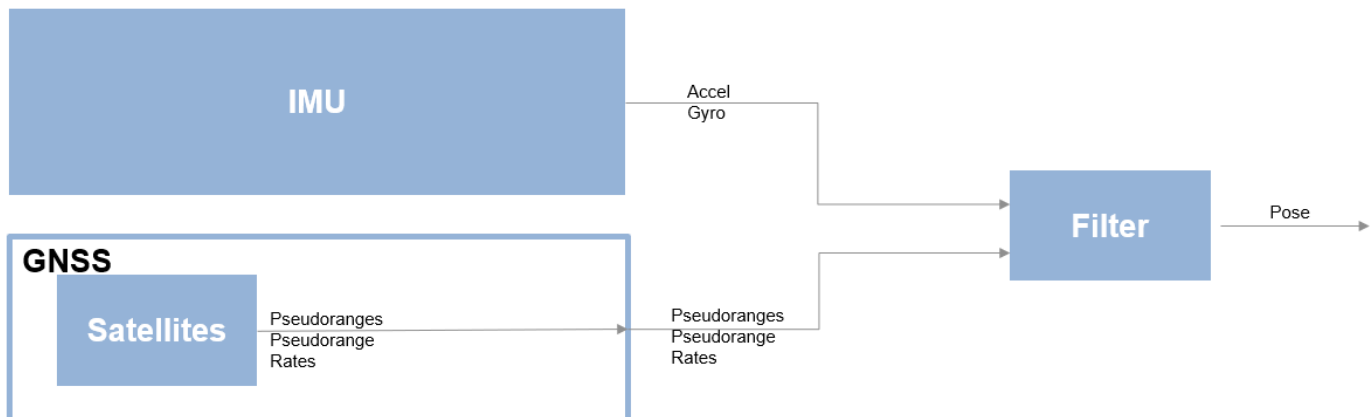
Ground Vehicle Pose Estimation for Tightly Coupled IMU and GNSS

This example shows how to estimate the position and orientation of a ground vehicle by building a tightly coupled extended Kalman filter and using it to fuse sensor measurements. A tightly coupled filter fuses inertial measurement unit (IMU) readings with raw global navigation satellite system (GNSS) readings. In contrast, a loosely coupled filter fuses IMU readings with filtered GNSS receiver readings.

Loosely Coupled Filter Diagram



Tightly Coupled Filter Diagram



Though a tightly coupled filter requires additional processing, you can use it when fewer than four GNSS satellite signals are available, or when some satellite signals are corrupted by interference such as multipath noise. This type of noise can occur when a ground vehicle is traveling through a portion of road with many obstructions, such as an urban canyon, where many surfaces reflect a combination of signals back into the receiver and interfere with the direct signal. Because the ground vehicle is in an urban environment with a lot of opportunity for multipath noise, this example uses a tightly coupled filter.

Define Filter Input

Specify these parameters for the sensor simulation:

- 1 Trajectory motion quantities
- 2 IMU sampling rate
- 3 GNSS receiver sampling rate
- 4 RINEX navigation message file

The RINEX navigation message file contains GNSS satellite orbital parameters used to compute satellite positions and velocities. The satellite positions and velocities are important for generating the GNSS pseudoranges and pseudorange rates.

```
load("routeNatickMATightlyCoupled","pos","orient","vel","acc","angvel","lla0","imuFs","gnssFs");
navfilename = "GODS00USA_R_20211750000_01D_GN.rnx";
```

Create the IMU sensor object. Set the axes misalignment and constant bias values to 0 to simulate calibration.

```
imu = imuSensor(SampleRate=imuFs);
loadparams(imu,"generic.json","GenericLowCost9Axis");
imu.Accelerometer.AxesMisalignment = 100*eye(3);
imu.Accelerometer.ConstantBias = [0 0 0];
imu.Gyroscope.AxesMisalignment = 100*eye(3);
imu.Gyroscope.ConstantBias = [0 0 0];
```

Read the RINEX file by using the `rinexread` function. Use only the first set of GPS satellites from the file, and set the initial time for the simulation to the first time step in the file.

```
data = rinexread(navfilename);
[~,idx] = unique(data.GPS.SatelliteID);
navmsg = data.GPS(idx,:);
t0 = navmsg.Time(1);
```

Load the noise parameters, `paramsTuned`, for the IMU sensor and GNSS receiver from the `tunedNoiseParameters` MAT file.

```
load("tunedNoiseParameters.mat","paramsTuned");
accelNoise = paramsTuned.AccelerometerNoise;
gyroNoise = paramsTuned.GyroscopeNoise;
pseudorangeNoise = paramsTuned.exampleHelperINSGNSSNoise(1);
pseudorangeRateNoise = paramsTuned.exampleHelperINSGNSSNoise(2);
```

Set the RNG seed to produce repeatable results.

```
rng default
```

Create Filter and Filter Sensor Models

To create the tightly coupled filter by using the `insEKF` object, you must define the conversion of filter states to raw GNSS measurements. Use the `exampleHelperINSGNSS` helper function to define this conversion.

```
gnss = exampleHelperINSGNSS;
gnss.ReferenceLocation = lla0;
```

Define an IMU model by creating accelerometer and gyroscope models using the `insAccelerometer` and `insGyroscope` objects, respectively.

```
accel = insAccelerometer;
gyro = insGyroscope;
```

Create the filter by using the IMU sensor model, the raw GNSS sensor model, and a 3-D pose motion model represented as an `insMotionPose` object.

```
filt = insEKF(accel,gyro,gnss,insMotionPose);
```

Set the initial states of the filter using tuned parameters from `paramsTuned`.

```
filt.State = paramsTuned.InitialState;
filt.StateCovariance = paramsTuned.InitialStateCovariance;
filt.AdditiveProcessNoise = paramsTuned.AdditiveProcessNoise;
```

Estimate Vehicle Pose

Create a figure in which to view the position estimate for the ground vehicle during the filtering process.

```
figure
posLLA = ned2lla(pos,lla0,"ellipsoid");
geoLine = geoplot(posLLA(1,1),posLLA(1,2),".",posLLA(1,1),posLLA(1,2),".");
geolimits([42.2948 42.3182],[-71.3901 -71.3519])
geobasemap topographic
legend("Ground truth","Filter estimate")
```

Allocate a matrix in which to store the position estimate results.

```
numSamples = size(pos,1);
estPos = NaN(numSamples,3);
estOrient = ones(numSamples,1,"quaternion");
imuSamplesPerGNSS = imuFs/gnssFs;
numGNSSSamples = numSamples/imuSamplesPerGNSS;
```

Set the current simulation time to the specified initial time.

```
t = t0;
```

Fuse the IMU and raw GNSS measurements. In each iteration, fuse the accelerometer and gyroscope measurements to the GNSS measurements separately to update the filter states, with the covariance matrices defined by the previously loaded noise parameters. After updating the filter state, log the new position and orientation states. Finally, predict the filter states to the next time step.

```
for ii = 1:numGNSSSamples
    for jj = 1:imuSamplesPerGNSS
        imuIdx = (ii-1)*imuSamplesPerGNSS + jj;
        [accelMeas,gyroMeas] = imu(acc(imuIdx,:),angvel(imuIdx,:),orient(imuIdx,:));

        fuse(filt,accel,accelMeas,accelNoise);
        fuse(filt,gyro,gyroMeas,gyroNoise);

        estPos(imuIdx,:) = stateparts(filt,"Position");
        estOrient(imuIdx,:) = quaternion(stateparts(filt,"Orientation"));

        t = t + seconds(1/imuFs);
```

```
        predict(filt,1/imuFs);  
end
```

Update the satellite positions and raw GNSS measurements.

```
gnssIdx = ii*imuSamplesPerGNSS;  
recPos = posLLA(gnssIdx,:);  
recVel = vel(gnssIdx,:);  
[satPos,satVel,satIDs] = gnssconstellation(t,"RINEXData",navmsg);  
[az,el,vis] = lookangles(recPos,satPos);  
[p,pdot] = pseudoranges(recPos,satPos(vis,:),recVel,satVel(vis,:));  
z = [p; pdot];
```

Update the satellite positions on the GNSS model.

```
gnss.SatellitePosition = satPos(vis,:);  
gnss.SatelliteVelocity = satVel(vis,:);
```

Fuse the raw GNSS measurements.

```
fuse(filt,gnss,z, ...  
    [pseudorangeNoise*ones(1,numel(p)),pseudorangeRateNoise*ones(1,numel(pdot))]);  
estPos(gnssIdx,:) = stateparts(filt,"Position");  
estOrient(gnssIdx,:) = quaternion(stateparts(filt,"Orientation"));
```

Update the position estimation plot.

```
estPosLLA = ned2lla(estPos,lla0,"ellipsoid");  
set(geoLine(1),LatitudeData=posLLA(1:gnssIdx,1),LongitudeData=posLLA(1:gnssIdx,2));  
set(geoLine(2),LatitudeData=estPosLLA(1:gnssIdx,1),LongitudeData=estPosLLA(1:gnssIdx,2));  
drawnow limitrate  
end
```



Validate Results

Calculate the RMS error to validate the results. The tightly coupled filter uses the provided sensor readings to estimate the ground vehicle pose with a relatively low RMS error.

```
posDiff = estPos - pos;
posRMS = sqrt(mean(posDiff.^2));
disp(['3-D Position RMS Error - X: ', num2str(posRMS(1)), ', Y:', ...
      num2str(posRMS(2)), ', Z: ', num2str(posRMS(3)), ' (m)'])
```

3-D Position RMS Error - X: 0.85213, Y:0.62864, Z: 0.9604 (m)

```
orientDiff = rad2deg(dist(estOrient,orient));
orientRMS = sqrt(mean(orientDiff.^2));
disp(['Orientation RMS Error - ', num2str(orientRMS), ' (degrees)'])
```

Orientation RMS Error - 4.077 (degrees)

Estimate Orientation Using GPS-Derived Yaw Measurements

This example shows how to define and use a custom sensor model for the `inSEKF` object along with built-in sensor models. Using a custom yaw angle sensor, an accelerometer, and a gyroscope, this example uses the `inSEKF` object to determine the orientation of a vehicle. You use the velocity from a GPS receiver to compute the yaw of the vehicle. Following a similar approach as shown in this example, you can develop custom sensor models for your own sensor fusion applications.

This example requires either the Sensor Fusion and Tracking Toolbox or the Navigation Toolbox. This example also optionally uses MATLAB Coder to accelerate filter tuning.

Problem Description

You are trying to estimate the orientation of a vehicle while it is moving. The only sensors available are an accelerometer, a gyroscope, and a GPS receiver that outputs a velocity estimate. You cannot use a magnetometer because there is a large amount of magnetic interference on the vehicle.

Approach

There are several tools available in the toolbox to determine orientation including

- `ecompass`
- `imufilter`
- `ahrsfilter`
- `complementaryFilter`

However, these filters require some combination of an accelerometer, a gyroscope, and/or a magnetometer. If you need to determine the absolute orientation (relative to True North) and do not have a magnetometer available, then none of these filters is ideal. The `imufilter` and `complementaryFilter` objects fuse accelerometer and gyroscope data, but they only provide a relative orientation. Furthermore, while these filters can estimate and compensate for the gyroscope bias, there is no additional sensor to help track the gyroscope bias for the yaw angle, which can yield a suboptimal estimate. Typically, a magnetometer is used to accomplish this. However, as mentioned, magnetometers cannot be used in situations with large, highly varying magnetic disturbances. (Note that the `ahrsfilter` object can handle mild magnetic disturbances over a short period of time).

In this example, to get the absolute orientation you use the GPS velocity estimate to determine the yaw angle. This yaw angle estimate can serve the same purpose as a magnetometer without the magnetic disturbance issues. You will build a custom sensor in the `inSEKF` framework to model this GPS-based raw sensor.

Trajectory

First, you create a ground truth trajectory and simulate the sensor by using the `exampleHelperMakeTrajectory` and `exampleHelperMakeIMUGPSData` functions attached with this example

```
groundTruth = exampleHelperMakeTrajectory;  
originalSensorData = exampleHelperMakeIMUGPSData(groundTruth);
```

Retuned as a timetable, the original sensor data includes the accelerometer, gyroscope, and GPS velocity data. Transform the GPS velocity data into a yaw angle estimate.

```
vel yaw = atan2(originalSensorData.GPSVelocity(:,2), originalSensorData.GPSVelocity(:,1));
```


The above results assume nonholonomic constraints. That is, they assume the vehicle is pointing in the direction of motion. This assumption is generally true some vehicles, such as a car, but not true for some other vehicles, such as a quadcopter.

Create Synthetic Yaw Angle Sensor Data

Yaw angle is difficult to work with because the yaw angle wraps at π and $-\pi$. The angle jumping at the wrapping bound could cause divergence of the extended Kalman filter. To avoid this problem, convert the yaw angle to a quaternion.

```
velyaw(:,2) = 0; % set pitch and roll to 0
velyaw(:,3) = 0;
qyaw = quaternion(velyaw, 'euler', 'ZYX', 'frame');
```

A common convention is to force the quaternion to have a positive real part

```
isNegQuat = parts(qyaw) < 0; % Find quaternions with a non-negative real part
qyaw(isNegQuat) = -qyaw(isNegQuat); % Invert the negative quaternions.
```

Note that when the `insEKF` object tracks a 4-element `Orientation` state as in this example, it assumes the `Orientation` state is a quaternion and enforces the quaternion to be normalized and have a positive real part. You can disable this rule by calling the state something else, like "Attitude".

Now you can build the timetable of the sensor data to be fused.

```
sensorData = removevars(originalSensorData, 'GPSVelocity');
sensorData = addvars(sensorData, compact(qyaw), 'NewVariableNames', 'YawSensor');
```

Create an `insEKF` Sensor for Fusing Yaw Angle

To fuse the yaw angle quaternion, you customize a sensor model and use it along with the `insAccelerometer` and `insGyroscope` objects in the `insEKF` object. To customize the sensor model, you create a class that inherits from the `positioning.INSSensorModel` interface class and implement the measurement method. Name the class `exampleHelperYawSensor`

```
classdef exampleHelperYawSensor < positioning.INSSensorModel
%EXAMPLEHELPERYAWSENSOR Yaw angle as quaternion sensor
% This class is for internal use only. It may be removed in the future.

% Copyright 2021 The MathWorks, Inc.

methods
function z = measurement(~, filt)
% Return an estimate of the measurement based on the
% state vector of the filter saved in filt.State.
%
% The measurement is just the quaternion converted from the yaw
% of the orientation estimate, assuming roll=0 and pitch=0.

q = quaternion(stateparts(filt, "Orientation"));
eul = euler(q, 'ZYX', 'frame');
yawquat = quaternion([eul(1) 0 0], 'euler', 'ZYX', 'frame');

% Enforce a positive quaternion convention
if parts(yawquat) < 0
    yawquat = -yawquat;
end
end
end
```

```
        end
        % Return a compacted quaternion
        z = compact(yawquat);
    end
end
end
```

Now use a `exampleHelperYawSensor` object alongside an `insAccelerometer` object and an `insGyroscope` object to construct the `insEKF` object.

```
opts = insOptions(SensorNamesSource='property', SensorNames={'Accelerometer', 'Gyroscope', 'YawSensor'});
filt = insEKF(insAccelerometer, insGyroscope, exampleHelperYawSensor, insMotionOrientation, opts);
```

Initialize the filter using the `stateparts` and `statecovparts` object functions.

```
stateparts(filt, 'Orientation', sensorData.YawSensor(1,:));
statecovparts(filt, 'Accelerometer_Bias', 1e-3);
statecovparts(filt, 'Gyroscope_Bias', 1e-3);
```

Filter Tuning

You can use the `tune` object function to find the optimal noise parameters for the `insEKF` object. You can directly call the `tune` object function, or you can use a MEX-accelerated cost function with MATLAB Coder.

```
% Trim ground truth to just contain the Orientation for tuning.
trimmedGroundTruth = timetable(groundTruth.Orientation, ...
    SampleRate=groundTruth.Properties.SampleRate, ...
    VariableNames={'Orientation'});

% Use MATLAB Coder to accelerate tuning by MEXing the cost function.
% To run the MATLAB Coder accelerated path, prior to running the example,
% type:
%   exampleHelperUseAcceleratedTuning(true);
% To avoid using MATLAB Coder, prior to the example, type:
%   exampleHelperUseAcceleratedTuning(false);
% By default, the example will not tune the filter live and will not use
% MATLAB Coder.

% Select the accelerated tuning option.
acceleratedTuning = exampleHelperUseAcceleratedTuning();

if acceleratedTuning
    createTunerCostTemplate(filt); % A new cost function in the editor
    % Save and close the file
    doc = matlab.desktop.editor.getActive;
    doc.saveAs(fullfile(pwd, 'tunercost.m'));
    doc.close;
    % Find the first parameter for codegen
    p = tunerCostFcnParam(filt); %#ok<NASGU>
    % Now generate a mex file
    codegen tunercost.m -args {p, sensorData, trimmedGroundTruth};
    % Use the Custom Cost Function and run the tuner for 20 iterations
    tunerIters = 20;
    cfg = tunerconfig(filt, MaxIterations=tunerIters, ...
        Cost='custom', CustomCostFcn=@tunercost_mex, ...
        StepForward=1.5, ...
```

```

ObjectiveLimit=0.0001, ...
FunctionTolerance=1e-6, ...
Display='none');

mnoise = tunernoise(filt);
tunedmn = tune(filt, mnoise, sensorData, trimmedGroundTruth, cfg);
else
% Use optimal parameters obtained separately.
tunedmn = struct(...
'AccelerometerNoise', 0.7786515625000000, ...
'GyroscopeNoise', 167.8674323338237, ...
'YawSensorNoise', 1.003122320344434);

adp = diag([...
1.2650000000000000;
1.181989791398048;
0.735171900658607;
0.7650000000000000;
0.026248409763699;
0.154586330266264;
31.823154516336434;
0.000546245218270;
5.517012554348883;
0.8090859375000000;
0.139035477206961;
41.340145917279159;
0.5928750000000000]);
filt.AdditiveProcessNoise = adp;
end

```

Fuse Data and Compare to Ground Truth

Batch fuse the data with the `estimateStates` object function.

```
est = estimateStates(filt, sensorData, tunedmn)
```

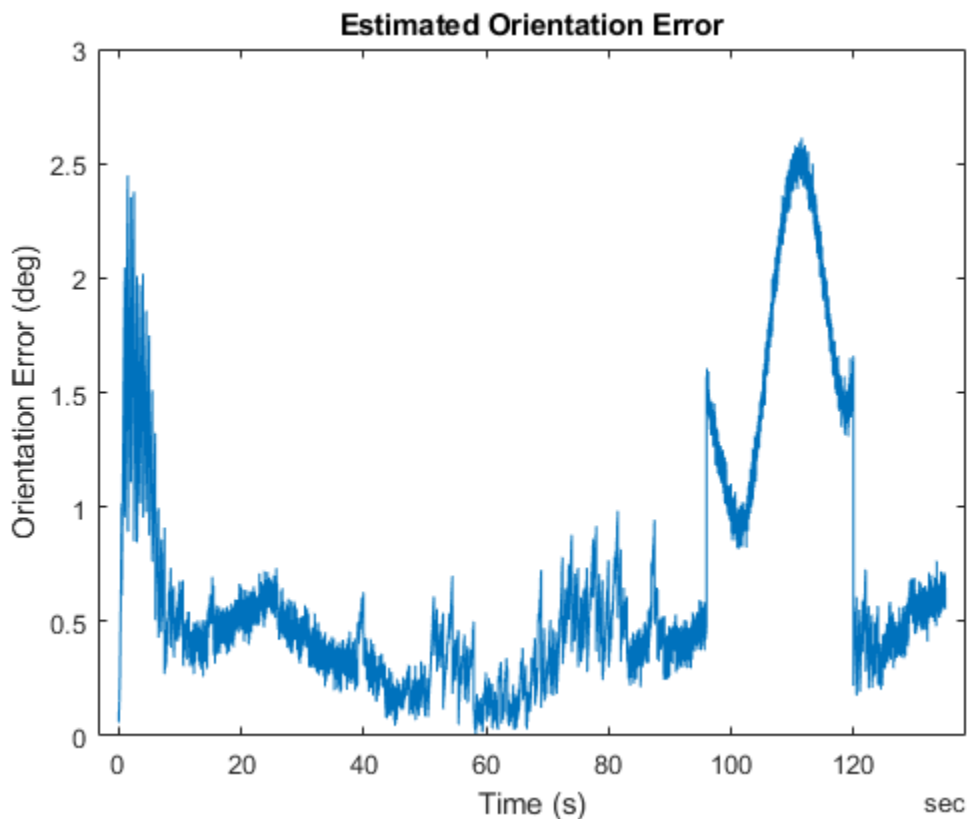
```
est=13500x5 timetable
```

Time	Orientation	AngularVelocity			Accelerom	
0 sec	1x1 quaternion	0.00034524	0.00030616	0.00029964	-1.2407e-08	-6.200
0.01 sec	1x1 quaternion	0.00037159	0.00035602	0.00071519	9.5542e-09	-0.00
0.02 sec	1x1 quaternion	0.00092989	0.00085552	0.0012341	-1.4108e-09	-1.33
0.03 sec	1x1 quaternion	0.0016774	0.001124	0.0017931	1.4709e-09	0.00
0.04 sec	1x1 quaternion	0.001733	0.0013256	0.0024162	7.0237e-09	9.29
0.05 sec	1x1 quaternion	0.0019492	0.0017904	0.0031902	-2.11e-09	5.13
0.06 sec	1x1 quaternion	0.0025321	0.0022415	0.0039831	-1.3683e-08	0.00
0.07 sec	1x1 quaternion	0.00252	0.0024167	0.0047767	-9.7718e-09	4.28
0.08 sec	1x1 quaternion	0.003163	0.0028615	0.0057521	-1.9926e-08	0.00
0.09 sec	1x1 quaternion	0.0032297	0.0026613	0.006758	5.1045e-09	0.00
0.1 sec	1x1 quaternion	0.0034874	0.0028447	0.0077077	6.8099e-09	0.00
0.11 sec	1x1 quaternion	0.0034133	0.0029448	0.00877	1.3428e-08	-0.00
0.12 sec	1x1 quaternion	0.0037193	0.0031047	0.0099134	1.7482e-08	-0.00
0.13 sec	1x1 quaternion	0.0041719	0.0036375	0.011024	-1.1761e-09	-1.11
0.14 sec	1x1 quaternion	0.0041973	0.0037822	0.012239	1.1026e-09	-0.00
0.15 sec	1x1 quaternion	0.0044736	0.0039896	0.013347	-4.9256e-10	-0.00
:						

Compare the estimated orientation to ground truth orientation. Plot the quaternion distance between the estimated and true orientations.

```
% Convert compacted quaternions to regular quaternions
estOrient = quaternion(est.Orientation);

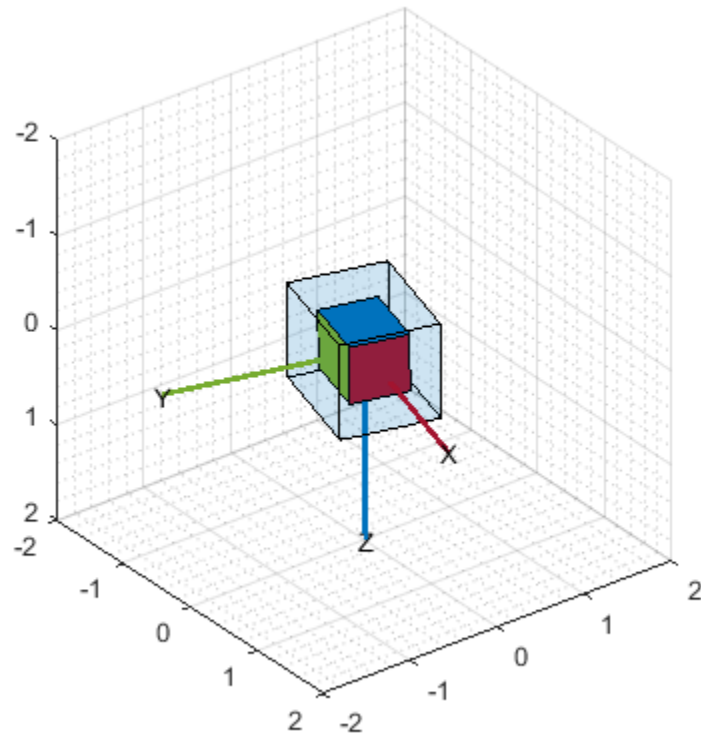
d = rad2deg(dist(estOrient, groundTruth.Orientation));
plot(groundTruth.Time, d);
xlabel('Time (s)')
ylabel('Orientation Error (deg)');
title('Estimated Orientation Error');
```



```
snapnow;
```

From the results, the estimated orientation error is low overall. There is a small bump in the error around 100 seconds, which is likely due to a slight inflection in the ground truth yaw angle at the time. But overall, this is a good orientation estimate result for many applications. You can visualize the estimated orientation using the `poseplot` function.

```
p = poseplot(estOrient(1));
for ii=2:numel(estOrient)
    p.Orientation = estOrient(ii);
    drawnow limitrate;
end
```



Conclusion

In this example, you learned how to customize a sensor and add it to the `insEKF` framework. Custom sensors can integrate with the built-in sensors like `insAccelerometer` and `insGyroscope`. You also learned how to use the `tune` object function to find optimal noise parameters and improve the filtering results.

Design Fusion Filter for Custom Sensors

This example introduces how to customize sensor models used with an `insEKF` object.

Using the `insEKF` object, you can fuse measurement data from multiple types of sensors by using the built-in INS sensor models, including the `insAccelerometer`, `insGyroSpace`, `insMagnetometer`, and `insGPS` objects. Though these sensor objects cover a variety of INS sensor models, you may want to fuse measurement data from a different type of sensor. Using the `insEKF` object through an extended framework, you can define your own sensor models and motion models used by the filter. In this example, you learn how to customize three sensor models in a few steps. You can apply the similar steps for defining a motion model.

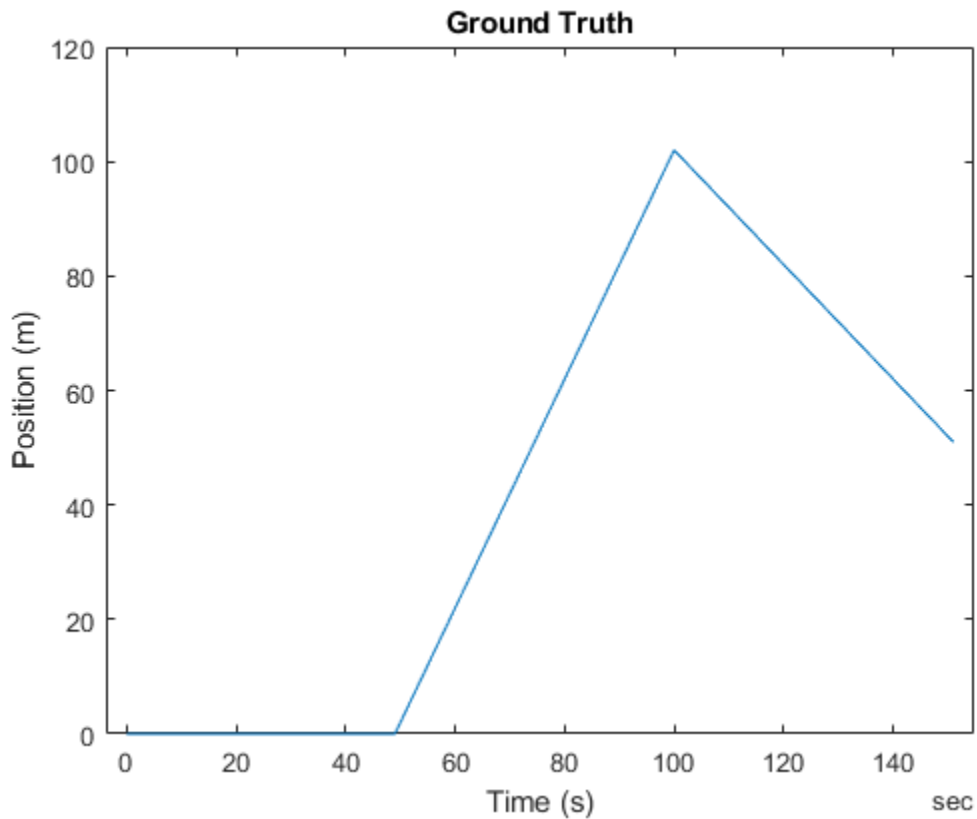
This example requires the Sensor Fusion and Tracking Toolbox or the Navigation Toolbox. This example also optionally uses MATLAB Coder to accelerate filter tuning.

Problem Description

Consider you are trying to estimate the position of an object that moves in one dimension. To estimate the position, you use a velocity sensor and fuse data from that sensor to determine the position. Typically, applying this approach can lead to a poor estimate of position because small errors in the velocity estimate can integrate to form larger errors in the position estimate. As shown later in this example, combining measurements from multiple sensors can improve the results.

To start, define a simple ground truth trajectory for the object and visualize how it moves.

```
groundTruth = exampleHelperMakeGroundTruth();  
plot(groundTruth.Time, groundTruth.Position);  
xlabel("Time (s)");  
ylabel("Position (m)");  
title("Ground Truth");
```



```
snapnow;
```

Simple Velocity Sensor

Consider the data from a simple velocity sensor that measures velocity but is corrupted by a small bias and additive white Gaussian noise.

```
velBias = exampleHelperVelocityWithBias(groundTruth);
```

For the simplest approach, you create a filter that treats the velocity measurements as just velocity plus white noise. This will likely not produce desirable results, but it is worth trying the simplest model first. To create a velocity sensor model for the `inSEKF` object, you need to define a class that inherits from the `positioning.INSSensorModel` interface class. At minimum you need to implement a `measurement` method which takes the sensor object and filter object as inputs. The `measurement` method returns a vector `z` as an estimate of measurement from the sensor, based on state variables.

```
classdef exampleHelperVelocitySensor < positioning.INSSensorModel
%EXAMPLEHELPERVELOCITYSENSOR A simple velocity sensor for inSEKF
% This class is for internal use only. It may be removed in the future.

% Copyright 2021 The MathWorks, Inc.

methods
function z = measurement(~, filt)
% The measurement is just the velocity estimate
z = stateparts(filt, 'Velocity');
```

```
        end
    end
end
```

This is the minimum required to define a sensor that works with the `insEKF` object. You could optionally define the measurement Jacobian in the `measurementJacobian` method, which can improve the calculation speed and possibly improve the estimation accuracy. Without implementing the `measurementJacobian` method, you are relying on the `insEKF` to numerically compute the Jacobian.

You build the filter with a simple 1-D constant velocity motion model defined using a `BasicConstantVelocityMotion` class. This class is an example of how to implement your own custom motion models by inheriting from the `positioning.INSMotionModel` interface class. Defining custom motion models requires implementing only a subset of the methods required to implement a custom sensor model.

You can now build the filter, tune its noise parameters, and see how it performs. You can name the Velocity sensor "VelocityWithBias" using the `insOptions` object.

```
basicOpts = insOptions(SensorNamesSource="property", SensorNames={'VelocityWithBias'});
basicfilt = insEKF(exampleHelperVelocitySensor, exampleHelperConstantVelocityMotion, ...
    basicOpts);
```

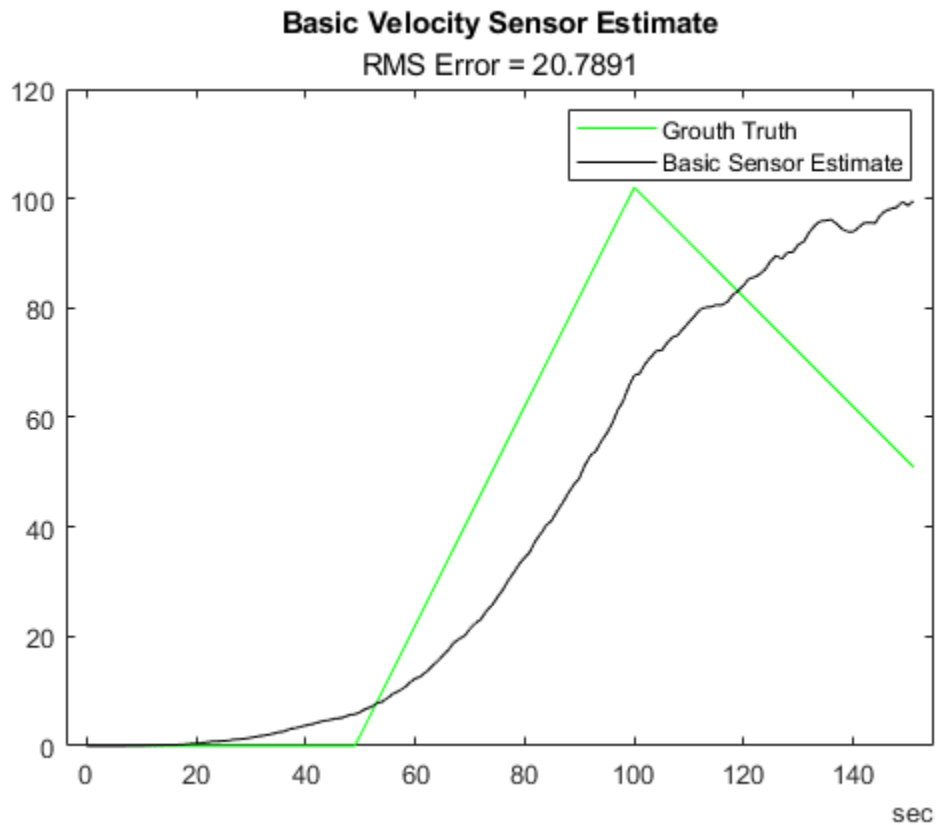
The object starts at rest with a position of zero. Set the state covariance for the position and velocity to a lower value before tuning.

```
statecovparts(basicfilt, "Position", 1e-2);
statecovparts(basicfilt, "Velocity", 1e-2);
```

Tune the filter after specifying a noise structure and a `tunerconfig` object.

```
mn = tunernoise(basicfilt);
tunerIters = 30;
cfg = tunerconfig(basicfilt, MaxIterations=tunerIters, Display='none');
tunedmn = tune(basicfilt, mn, velBias, groundTruth, cfg);

estBasic = estimateStates(basicfilt, velBias, tunedmn);
figure;
plot(groundTruth.Time, groundTruth.Position, 'g', estBasic.Time, ...
    estBasic.Position, 'k')
title("Basic Velocity Sensor Estimate");
err = sqrt(mean((estBasic.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);
legend("Grouth Truth", "Basic Sensor Estimate");
```

```
snapnow;
```

The estimation results are poor as expected since the bias is not accounted for by the filter. When the unaccounted velocity bias is integrated into position, the estimate diverges quickly from ground truth.

Velocity Sensor With Bias

Now you can explore a more complex sensor model which accounts for the sensor bias. To do this, you define a `sensorstates` method to inform the `insEKF` to track the sensor bias in its state vector, saved in the `State` property of the filter. You can query the sensor bias with the `stateparts` object function of the filter and use the bias estimate in the `measurement` function.

```
classdef exampleHelperVelocitySensorWithBias < positioning.INSSensorModel
%EXAMPLEHELPERVELOCITYSENSORWITHBIAS Velocity sensor assuming constant bias
% This class is for internal use only. It may be removed in the future.
```

```
% Copyright 2021 The MathWorks, Inc.
```

```
methods
```

```
function s = sensorstates(~,~)
% Assume there is a constant bias corrupting the velocity
% measurement. Estimate that bias as part of the filter
% computation.
s = struct('Bias', 0);
```

```
end
```

```
function z = measurement(sensor, filt)
```

```
% Measurement is velocity plus bias. Obtain the velocity from
% the filter state.
velocity = stateparts(filt, 'Velocity');

% Obtain the sensor bias from the filter state by directly
% using the sensor object as an input to the stateparts object
% function. In this way, knowledge of the SensorName associated
% with this sensor is not required. See the reference page for
% stateparts for more details.
bias = stateparts(filt, sensor, 'Bias');

% Form the measurement
z = velocity + bias;
end
function dzdx = measurementJacobian(sensor, filt)
% Compute the Jacobian as the partial derivatives of the Bias
% state relative to all other states.
N = numel(filt.State); % Number of states

% Initialize a matrix of partial derivatives. This matrix has
% one row because the sensor state is a scalar.
dzdx = zeros(1,N);

% The partial derivative of z with respect to the Bias is 1. So
% put a 1 at the Bias index in the dzdx matrix:
bidx = stateinfo(filt, sensor, 'Bias');
dzdx(:,bidx) = 1;

% The partial derivative of z with respect to the Velocity is
% also 1. So put a 1 at the Bias index in the dzdx matrix:
vidx = stateinfo(filt, 'Velocity');
dzdx(:,vidx) = 1;
end
end
end
```

In the above you implemented the `measurementJacobian` method rather than relying on a numeric Jacobian. The Jacobian matrix contains the partial derivatives of the sensor measurement with respect to the filter state, which is an M -by- N matrix, where M is the number of elements in the vector returned by the measurement method and N is the number of states of the filter. Now you can build a filter with the custom sensor.

```
sensorVelWithBias = exampleHelperVelocitySensorWithBias;
velWithBiasFilt = insEKF(sensorVelWithBias, ...
    exampleHelperConstantVelocityMotion, basicOpts);
```

You can set the initial bias state of the filter with an estimate of the bias, based on data collected while the sensor is stationary, using the `stateparts` object function.

```
stationaryLen = 1e7;
stationary = timetable(zeros(stationaryLen,1), zeros(stationaryLen,1), ...
    TimeStep=groundTruth.Properties.TimeStep, ...
    VariableNames={'Position', 'Velocity'});
biasCalibrationData = exampleHelperVelocityWithBias(stationary);

stateparts(velWithBiasFilt, sensorVelWithBias, "Bias", ...
    mean(biasCalibrationData.VelocityWithBias));
```

Again, set the state covariance for these states to smaller values before tuning.

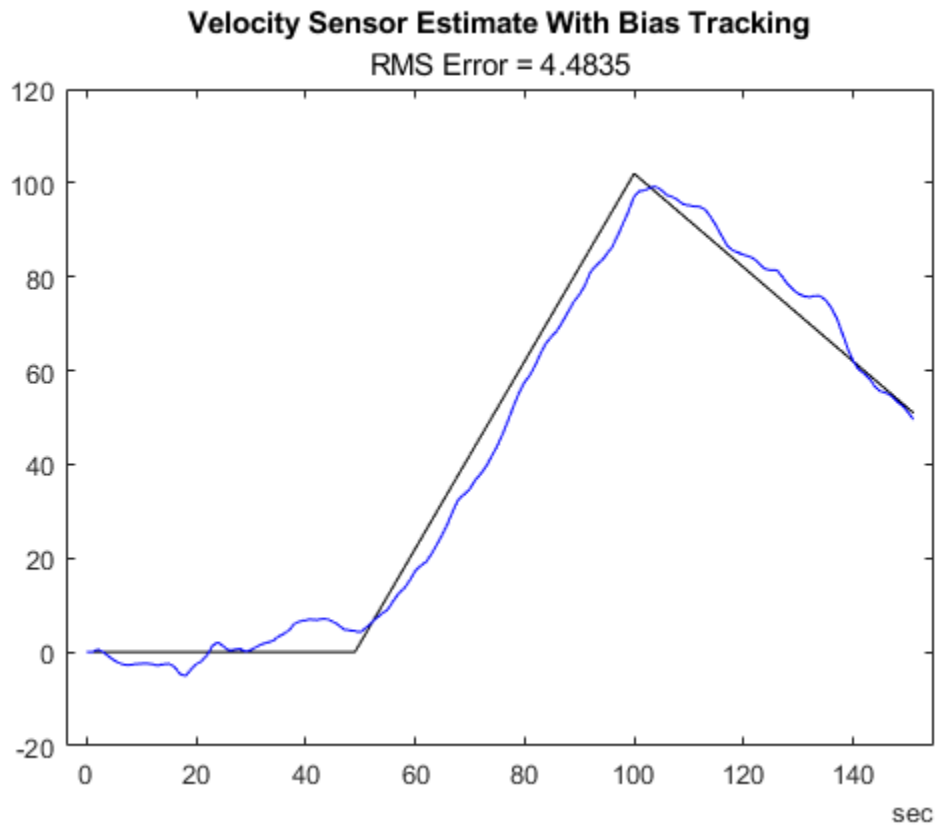
```
statecovparts(velWithBiasFilt, "Position", 1e-2);
statecovparts(velWithBiasFilt, "Velocity", 1e-2);
```

You can tune this filter as previously, but you can also tune it faster using MATLAB Coder. While the `tune` object function does not support code generation, you can use a MEX-accelerated cost function to greatly increase the tuning speed. You can specify the `tune` function to use a custom cost function through the `tunerconfig` input. The `inSEKF` object has object functions to help create a custom cost function. The `createTunerCostTemplate` object function creates a cost function, which tries to minimize the RMS error of state estimates, in a new document in the Editor. You can then generate code for that function with the help of the `tunerCostFcnParam` object function, which creates an example of the first argument to a cost function.

```
% Use MATLAB Coder to accelerate tuning by MEXing the cost function.
% To run the MATLAB Coder accelerated path, prior to running the example,
% type:
%   exampleHelperUseCodegenForCost(true);
% To avoid using MATLAB Coder, prior to the example, type:
%   exampleHelperUseCodegenForCost(false);

useCodegenForTuning = exampleHelperUseCodegenForCost();

if useCodegenForTuning
    createTunerCostTemplate(velWithBiasFilt); % A new cost function in the editor
    exampleHelperSaveCostFunction;
    p = tunerCostFcnParam(velWithBiasFilt);
    % Now generate a mex file
    codegen tunercost.m -args {p, velBias, groundTruth};
    % Use the Custom Cost Function
    cfg2 = tunerconfig(velWithBiasFilt, MaxIterations=tunerIters, ...
        Cost="custom", CustomCostFcn=@tunercost_mex, Display='none');
else
    % Use the default cost function
    cfg2 = tunerconfig(velWithBiasFilt, MaxIterations=tunerIters, ...
        Display='none'); %#ok<*UNRCH>
end
mn = tunernoise(velWithBiasFilt);
tunedmn = tune(velWithBiasFilt, mn, velBias, groundTruth, cfg2);
estWithBias = estimateStates(velWithBiasFilt, velBias, tunedmn);
figure;
plot(groundTruth.Time, groundTruth.Position, 'k', estWithBias.Time, ...
    estWithBias.Position, 'b')
title("Velocity Sensor Estimate With Bias Tracking");
err = sqrt(mean((estWithBias.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);
```



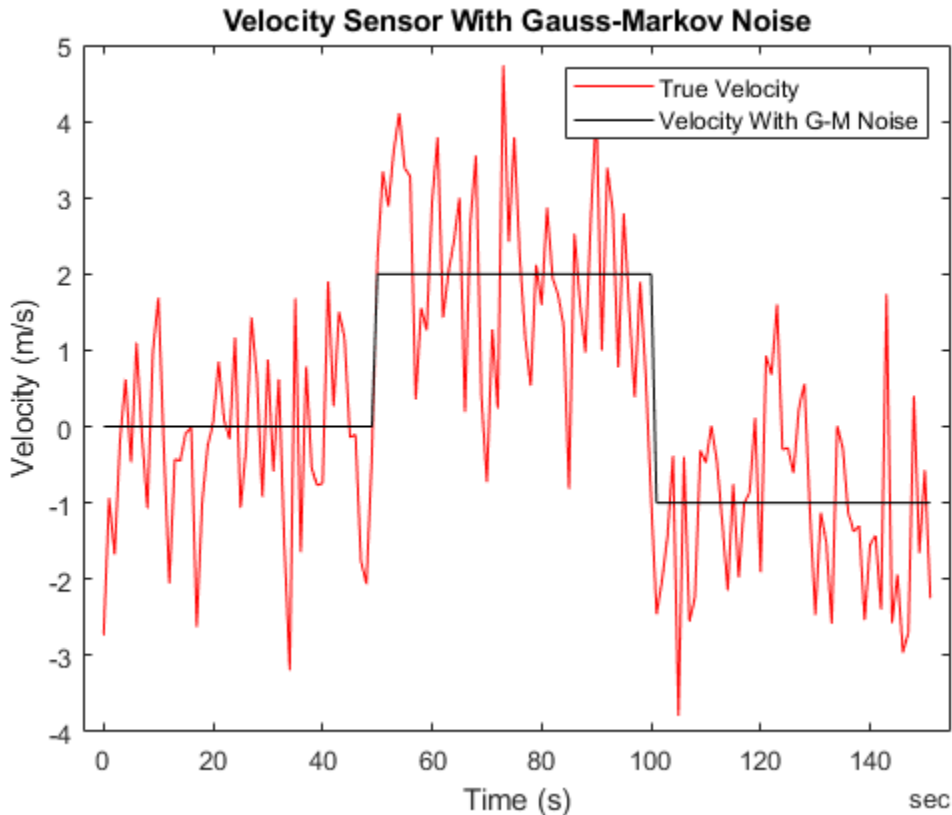
```
snapnow;
```

From the results, the position estimate has improved but is still not ideal. The filter estimates the bias, but any error in the bias estimate is treated as real velocity, which corrupts the position estimate. A more sophisticated sensor model can further improve the results.

Velocity Sensor with Gauss-Markov Noise

Consider a new velocity sensor model that is corrupted with first-order Gauss-Markov noise.

```
velGM = exampleHelperVelocityWithGaussMarkov(groundTruth);
figure
plot(velGM.Time, velGM.VelocityWithGM, 'r', groundTruth.Time, ...
     groundTruth.Velocity, 'k');
title("Velocity Sensor With Gauss-Markov Noise");
xlabel("Time (s)");
ylabel("Velocity (m/s)");
legend("True Velocity", "Velocity With G-M Noise");
```



snapshot;

First-order Gauss-Markov noise can be modeled in discrete time as:

$$x_k = (1 - \beta \cdot dt)x_{k-1} + w_k$$

where w_k is white noise. The equivalent continuous time formulation is:

$$\frac{d}{dt}x = -\beta x(t) + w(t)$$

where β is the time constant of the process, and $w(t)$ and $x(t)$ are the continuous time versions of the discrete state x_k and process noise w_k , respectively.

You can check that these are a valid discrete-continuous pair by applying Euler integration on the second equation between time k and $k-1$ to obtain the first equation. See Reference [1] for a further explanation. For simplicity, set β as 0.002 for this example. Since the Gauss-Markov process evolves over time you need to implement it in the `stateTransition` method of the `positioning.INSSensorModel` class. The `stateTransition` method describes how state elements described in `sensorstates` evolve over time. The `stateTransition` function outputs a structure with the same fields as the output structure of the `sensorstates` method, and each field contains the derivative of the state with respect to time. When the `stateTransition` method is not implemented explicitly, the `insEKF` object assumes the state is constant over time.

```
classdef exampleHelperVelocitySensorWithGM < positioning.INSSensorModel
%EXAMPLEHELPERVELOCITYSENSORWITHGM Velocity sensor with Gauss-Markov noise
```

```
% This class is for internal use only. It may be removed in the future.

% Copyright 2021 The MathWorks, Inc.

properties (Constant)
    Beta =0.002; % First-order Gauss-Markov time constant
end
methods
    function s = sensorstates(~,~)
        % Assume the velocity measurement is corrupted by a first-order
        % Gauss-Markov random process. Estimate the state of the
        % Gauss-Markov process as part of the filtering.
        s = struct('GMPProc', 0);
    end
    function z = measurement(sensor, filt)
        % Measurement of velocity plus Gauss-Markov process noise
        velocity = stateparts(filt, 'Velocity');
        gm = stateparts(filt, sensor, 'GMPProc');

        z = velocity + gm;
    end
    function dhdx = measurementJacobian(sensor, filt)
        % Compute the Jacobian of partial derivatives of the measurement
        % relative to all states.
        N = numel(filt.State); % Number of states

        % Initialize a matrix of partial derivatives. The matrix only
        % has one row because the sensor state is a scalar.
        dhdx = zeros(1,N);

        % Get the indices of the Velocity and GMPProc states in the
        % state vector.
        vidx = stateinfo(filt, 'Velocity');
        gmidx = stateinfo(filt, sensor, 'GMPProc');
        dhdx(:,gmidx) = 1;
        dhdx(:,vidx) = 1;
    end
    function sdot = stateTransition(sensor, filt, ~, varargin)
        % Define the state transition function for each sensorstates
        % defined in this class. Since the insEKF is a
        % continuous-discrete EKF, the output is the derivative of the
        % state in the form of a structure with field names matching
        % the field names of the output structure of sensorstates. The
        % first-order Gauss-Markov process has a state transition of
        %     sdot = -1*beta*s
        % where beta is the reciprocal of the time constant of the
        % process.
        sdot.GMPProc = -1*(sensor.Beta) * ...
            stateparts(filt, sensor, 'GMPProc');
    end
    function dfdx = stateTransitionJacobian(sensor, filt, ~, varargin)
        % Compute the Jacobian of the partial derivatives of the GMPProc
        % state transition relative to all states.

        % Find the number of states and initialize an array of the same
        % size with all elements equal to zero.
        N = numel(filt.State);
        dfdx.GMPProc = zeros(1,N);
    end
end
```

```

    % Find the index of the Gauss-Markov state, GMProc, in the
    % state vector.
    gmidx = stateinfo(filt, sensor, 'GMProc');

    % The derivative of the GMProc stateTransition function with
    % respect to GMProc is -1*Beta
    dfdx.GMProc(gmidx) = -1*(sensor.Beta);
end
end
end

```

Note here a `stateTransitionJacobian` method is also implemented. The method outputs a structure with its fields containing the partial derivatives of `sensorstates` with respect to the filter state vector. If this function is not implemented, the `insEKF` computes these matrices numerically. You can try commenting out this function to see the effects of using the numeric Jacobian.

```

gmOpts = insOptions(SensorNamesSource="property", SensorNames={'VelocityWithGM'});
filtWithGM = insEKF(exampleHelperVelocitySensorWithGM, ...
    exampleHelperConstantVelocityMotion, gmOpts);

```

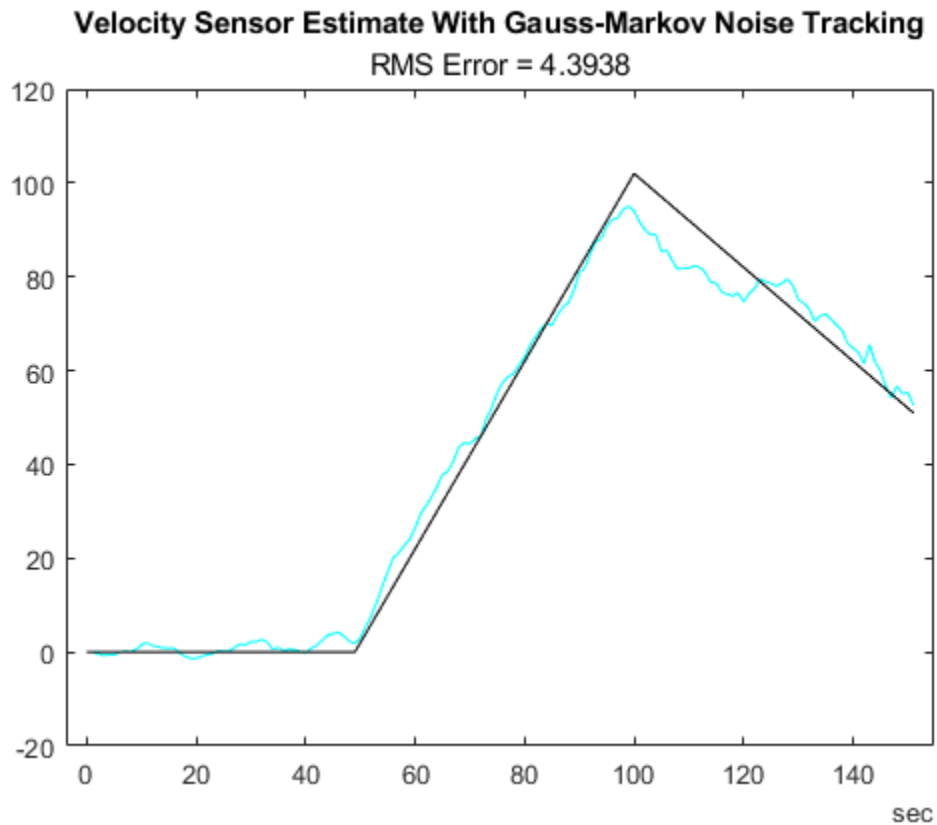
You can re-tune the filter following any of the processes above. Estimate the state by using the `estimateStates` object function.

```

statecovparts(filtWithGM, "Position", 1e-2);
statecovparts(filtWithGM, "Velocity", 1e-2);
gmMeasNoise = tunernoise(filtWithGM);
cfg3 = tunerconfig(filtWithGM, MaxIterations=tunerIters, Display='none');
gmTunedNoise = tune(filtWithGM, gmMeasNoise, velGM, groundTruth, cfg3);

estGM = estimateStates(filtWithGM, velGM, gmTunedNoise);
figure
plot(estGM.Time, estGM.Position, 'c', groundTruth.Time, ...
    groundTruth.Position, 'k');
title("Velocity Sensor Estimate With Gauss-Markov Noise Tracking");
err = sqrt(mean((estGM.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);

```



```
snapnow;
```

The filter again cannot distinguish between the Gauss-Markov noise and the true velocity. The Gauss-Markov noise is not observable independently and thus corrupts the position estimate. However, you can reuse the sensors designed above with measurement data from multiple sensors to get a better position estimate.

Multi-sensor fusion

To get an accurate estimate of position, you use multiple sensors. You apply the sensor models you have already designed in a new `insEKF` filter object.

```
combinedOpts = insOptions(SensorNamesSource="property", SensorNames={'VelocityWithBias', 'VelocityWithGM'});
sensorWithBias = exampleHelperVelocitySensorWithBias;
sensorWithGM = exampleHelperVelocitySensorWithGM;
filtCombined = insEKF(sensorWithBias, sensorWithGM, ...
    exampleHelperConstantVelocityMotion, combinedOpts);

% Combine the two sets of sensor measurements.
sensorData = synchronize(velBias, velGM, "first");
% Initialize the bias state with an estimate.
stateparts(filtCombined, sensorWithBias, "Bias", ...
    mean(biasCalibrationData.VelocityWithBias));

mnCombined = tunernoise(filtCombined);
cfg4 = tunerconfig(filtCombined, MaxIterations=tunerIters, Display='none');
combinedTunedNoise = tune(filtCombined, mnCombined, sensorData, ...
```

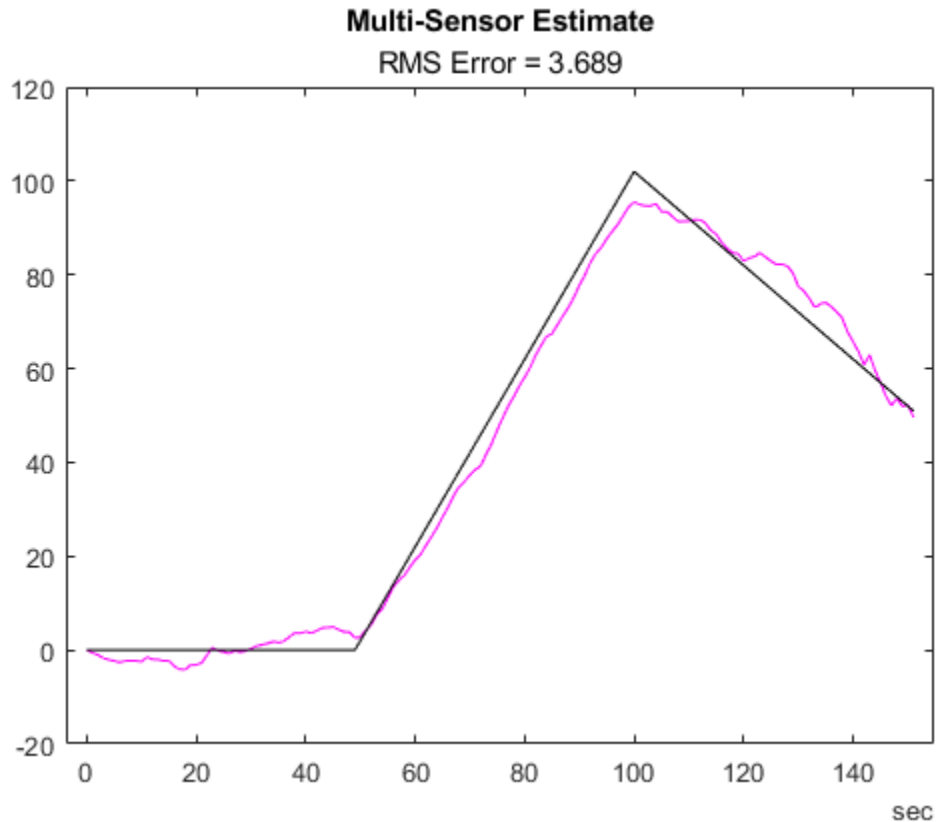


```

    groundTruth, cfg4);

estBoth = estimateStates(filtCombined, sensorData, combinedTunedNoise);
figure;
plot(estBoth.Time, estBoth.Position, 'm', groundTruth.Time, ...
     groundTruth.Position, 'k');
title("Multi-Sensor Estimate");
err = sqrt(mean((estBoth.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);

```



```

snapnow;

```

Because the filter uses two sensors, both the bias and Gauss-Markov noise are observable. As a result, the filter obtains a more accurate velocity estimate and thus more accurate position estimate.

Conclusion

In this example, you learned the `insEKF` framework and how to customize sensor models used with the `insEKF` object. Implementing a custom motion model is almost the same process as implementing a new sensor, except that you inherit from the `positioning.INSMotionModel` interface class instead of the `positioning.INSSensorModel` interface class. Designing custom sensor fusion filters is straightforward with the `insEKF` framework and it is simple to build reusable sensors to use across projects.

References

[1] A Comparison between Different Error Modeling of MEMS Applied to GPS/INS Integrated Systems. A. Quinchia, G. Falco, E. Falletti, F. DAVIS, C. Ferrer, Sensors 2013.

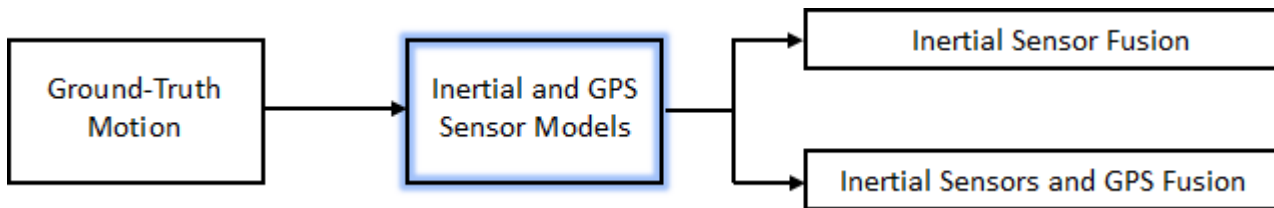
Navigation Topics

- “Model IMU, GPS, and INS/GPS” on page 2-2
- “Choose Inertial Sensor Fusion Filters” on page 2-8
- “Configure Time Scope MATLAB Object” on page 2-13
- “Fuse Inertial Sensor Data Using insEKF-Based Flexible Fusion Framework” on page 2-33
- “Occupancy Grids” on page 2-45
- “Choose Path Planning Algorithms for Navigation” on page 2-53
- “Execute Code at a Fixed-Rate” on page 2-55
- “Particle Filter Workflow” on page 2-58
- “Particle Filter Parameters” on page 2-62
- “Pure Pursuit Controller” on page 2-67
- “Monte Carlo Localization Algorithm” on page 2-69
- “Vector Field Histogram” on page 2-78

Model IMU, GPS, and INS/GPS

Navigation Toolbox enables you to model inertial measurement units (IMU), Global Positioning Systems (GPS), and inertial navigation systems (INS). You can model specific hardware by setting properties of your models to values from hardware datasheets. You can tune environmental and noise properties to mimic real-world environments. You can use these models to test and validate your fusion algorithms or as placeholders while developing larger applications.

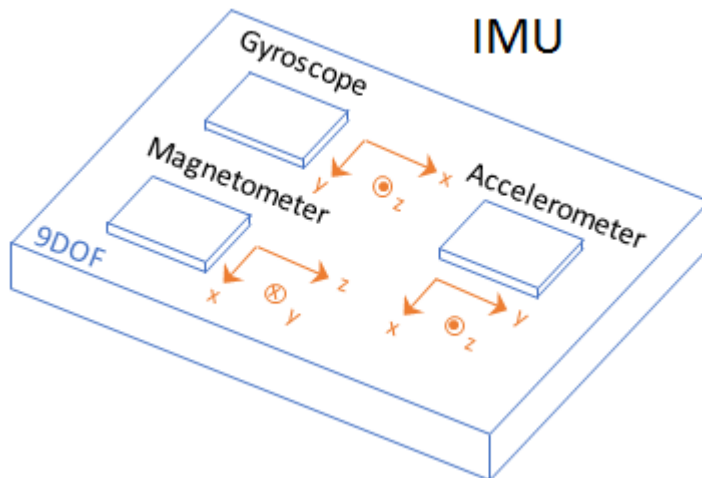
This tutorial provides an overview of inertial sensor and GPS models in Navigation Toolbox.



To learn how to generate the ground-truth motion that drives the sensor models, see `waypointTrajectory` and `kinematicTrajectory`.

Inertial Measurement Unit

An IMU is an electronic device mounted on a platform. The IMU consists of individual sensors that report various information about the platform's motion. IMUs combine multiple sensors, which can include accelerometers, gyroscopes, and magnetometers.



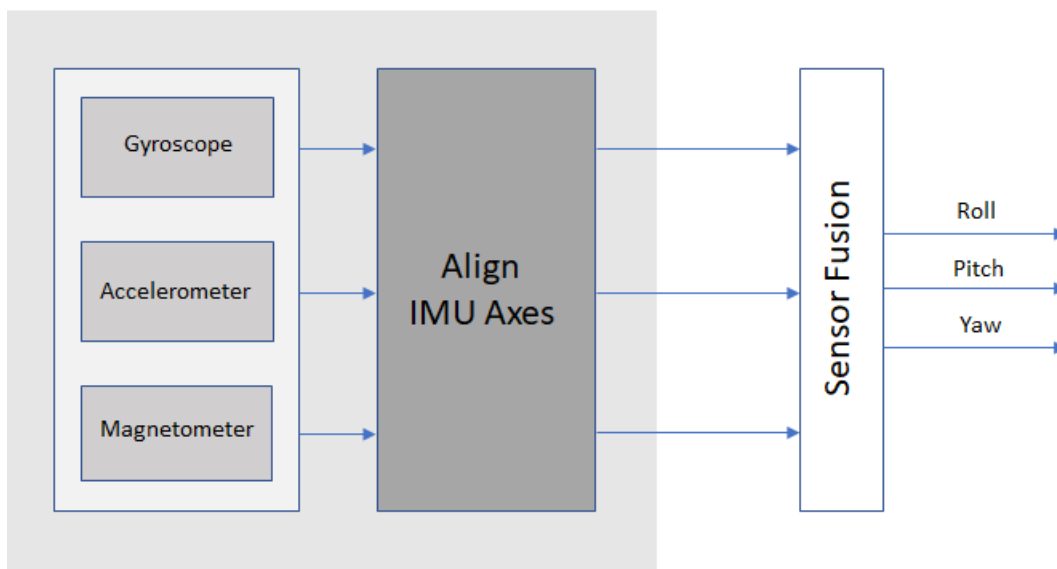
With this toolbox, measurements returned from an IMU model use the following unit and coordinate conventions.

Output	Description	Units	Coordinate System
Acceleration	Current accelerometer reading	m/s ²	Sensor Body

Output	Description	Units	Coordinate System
Angular velocity	Current gyroscope reading	rad/s	Sensor Body
Magnetic field	Current magnetometer reading	μT	Sensor Body

Usually, the data returned by IMUs is fused together and interpreted as roll, pitch, and yaw of the platform. Real-world IMU sensors can have different axes for each of the individual sensors. The models provided by Navigation Toolbox assume that the individual sensor axes are aligned.

IMU Model



To create an IMU sensor model, use the `imuSensor` System object™.

```
IMU = imuSensor
```

```
IMU =
```

```
imuSensor with properties:
```

```

    IMUType: 'accel-gyro'
    SampleRate: 100
    Temperature: 25
    Accelerometer: [1x1 accelparams]
    Gyroscope: [1x1 gyroparams]
    RandomStream: 'Global stream'

```

The default IMU model contains an ideal accelerometer and an ideal gyroscope. The `accelparams` and `gyroparams` objects define the accelerometer and gyroscope configuration. You can set the properties of these objects to mimic specific hardware and environments. For more information on IMU parameter objects, see `accelparams`, `gyroparams`, and `magparams`.

To model receiving IMU sensor data, call the IMU model with the ground-truth acceleration and angular velocity of the platform:

```
trueAcceleration = [1 0 0];  
trueAngularVelocity = [1 0 0];  
[accelerometerReadings,gyroscopeReadings] = IMU(trueAcceleration,trueAngularVelocity)
```

```
accelerometerReadings =
```

```
    -1.0000         0     9.8100
```

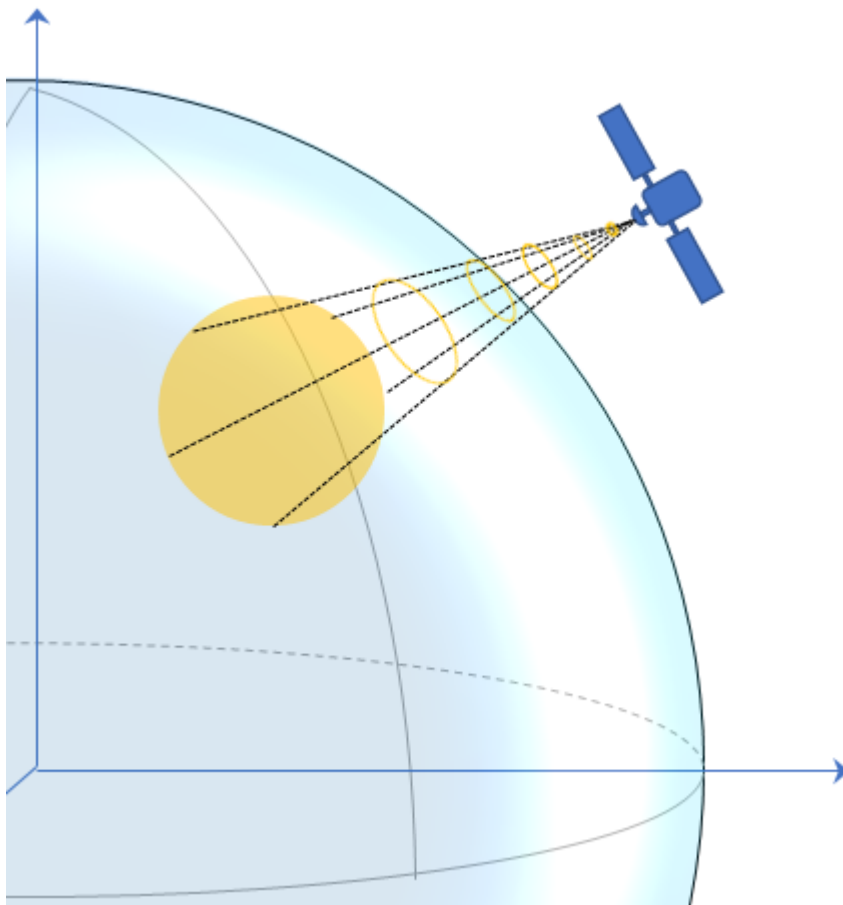
```
gyroscopeReadings =
```

```
     1     0     0
```

You can generate the ground-truth trajectories that you input to the IMU model using `kinematicTrajectory` and `waypointTrajectory`.

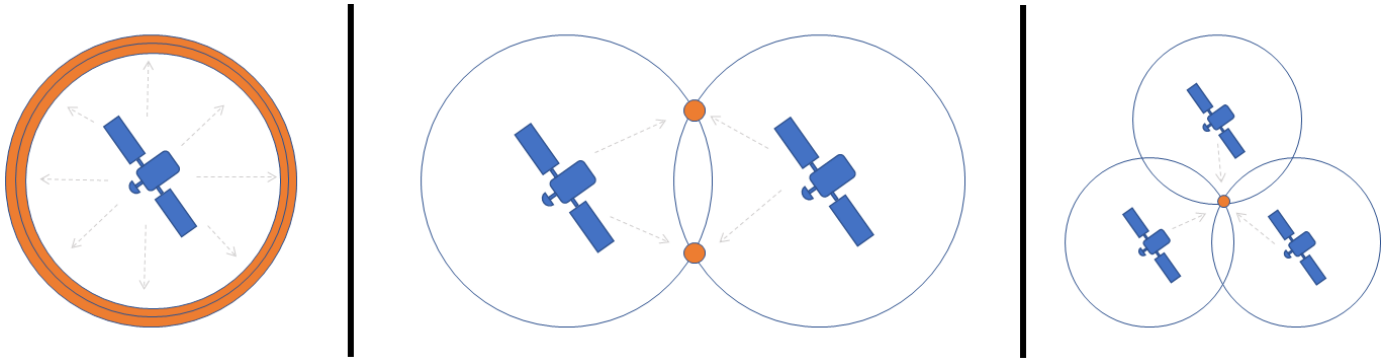
Global Positioning System

A global positioning system (GPS) provides 3-D position information for platforms (receivers) on the surface of the Earth.



GPS consists of a constellation of satellites that continuously orbit the earth. The satellites maintain a configuration such that a platform is always within view of at least four satellites. By measuring the flight time of signals from the satellites to the platform, the position of the platform can be

trilaterated. Satellites timestamp a broadcast signal, which is compared to the platform's clock upon receipt. Three satellites are required to trilaterate a position in three dimensions. The fourth satellite is required to correct for clock synchronization errors between the platform and satellites.



The GPS simulation provided by Navigation Toolbox models the platform (receiver) data that has already been processed and interpreted as altitude, latitude, longitude, velocity, groundspeed, and course.

Measurements returned from the GPS model use the following unit and coordinate conventions.

Output	Description	Units	Coordinate System
LLA	Current global position reading in geodetic coordinates, based on wgs84Ellipsoid Earth model	degrees (latitude), degrees (longitude), meters (altitude)	LLA
Velocity	Current velocity reading from GPS	m/s	local NED
Groundspeed	Current groundspeed reading from GPS	m/s	local NED
Course	Current course reading from GPS	degrees	local NED

The GPS model enables you to set high-level accuracy and noise parameters, as well as the receiver update rate and a reference location.

To create a GPS model, use the `gpsSensor` System object.

```
GPS = gpsSensor
```

```
GPS =
```

```
gpsSensor with properties:
```

```

    UpdateRate: 1                Hz
    ReferenceLocation: [0 0 0]    [deg deg m]
    HorizontalPositionAccuracy: 1.6 m
    VerticalPositionAccuracy: 3   m
    VelocityAccuracy: 0.1        m/s
    RandomStream: 'Global stream'
    DecayFactor: 0.999

```

To model receiving GPS sensor data, call the GPS model with the ground-truth position and velocity of the platform:

```
truePosition = [1 0 0];
trueVelocity = [1 0 0];
[LLA,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

```
LLA =
    0.0000    0.0000    0.3031
```

```
velocity =
    1.0919   -0.0008   -0.1308
```

```
groundspeed =
    1.0919
```

```
course =
    359.9566
```

You can generate the ground-truth trajectories that you input to the GPS model using `kinematicTrajectory` and `waypointTrajectory`.

Inertial Navigation System and Global Positioning System

An inertial navigation system (INS) uses inertial sensors like those found on an IMU: accelerometers, gyroscopes, and magnetometers. An INS fuses the inertial sensor data to calculate position, orientation, and velocity of a platform. An INS/GPS uses GPS data to correct the INS. Typically, the INS and GPS readings are fused with an extended Kalman filter, where the INS readings are used in the prediction step, and the GPS readings are used in the update step. A common use for INS/GPS is dead-reckoning when the GPS signal is unreliable.

"INS/GPS" refers to the entire system, including the filtering. The INS/GPS simulation provided by Navigation Toolbox models an INS/GPS and returns the position, velocity, and orientation reported by the inertial sensors and GPS receiver based on a ground-truth motion.

Measurements returned from the INS/GPS use the following unit and coordinate conventions.

Output	Description	Units	Coordinate System
Position	Current position reading from the INS/GPS	meters	local NED
Velocity	Current velocity reading from the INS/GPS	m/s	local NED
Orientation	Current orientation reading from the INS/GPS	quaternion or rotation matrix	N/A

See Also

`imuSensor` | `gpsSensor` | `insSensor` | `gnssSensor`

Related Examples

- “Introduction to Simulating IMU Measurements” on page 1-218
- “GNSS Simulation Overview” on page 1-85

External Websites

- <https://www.gps.gov/systems/gps/>

Choose Inertial Sensor Fusion Filters

The toolbox provides multiple filters to estimate the pose and velocity of platforms by using on-board inertial sensors (including accelerometer, gyroscope, and altimeter), magnetometer, GPS, and visual odometry measurements. Each filter can process certain types of measurements from certain sensors. Each filter also makes assumptions and may have limitations that you should consider carefully before applying it. For example, many filters assume no sustained linear or angular acceleration other than the gravitational acceleration. Therefore, you should avoid using them during strong and constant acceleration, but these filters can perform reasonably well during short linear acceleration bursts. Also, some filters allow piecewise constant linear acceleration and angular velocity since they allow acceleration and angular velocity inputs during the prediction step.

The internal algorithms of these filters also vary greatly. For example, the `ecompass` object uses the TRIAD method to determine the orientation of the platform with very low computation cost. Many filters (such as `ahrsfilter` and `imufilter`) adopt the error-state Kalman filter, in which the state deviation from the reference state is estimated. Meanwhile, other filters (such as `insfilterMARG` and `insfilterAsync`) use the extended Kalman filter approach, in which the state is estimated directly.

To achieve high estimation accuracy, it is important to tune the filter properties and parameters properly. The toolbox offers the built-in `tune` function to tune parameters and sensor noise for most of the inertial sensor filters (marked as tunable in the table below).

The table lists the inputs, outputs, assumptions, and algorithms for all the inertial sensor fusion filters.

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
<code>ecompass</code>	<ul style="list-style-type: none"> Accelerometer Magnetometer 	Orientation	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	TRIAD method	No
<code>ahrsfilter</code>	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer 	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Error-state Kalman filter	Yes

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
ahrs10filter	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer Altimeter 	Orientation, altitude, vertical velocity, delta angle bias, delta velocity bias, geomagnetic field vector, magnetometer bias	The filter assumes piecewise constant linear acceleration in the vertical direction, and no sustained linear and angular acceleration other than gravitational acceleration in other directions.	Discrete extended Kalman filter	Yes
imufilter	<ul style="list-style-type: none"> Accelerometer Gyroscope 	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Error-state Kalman filter	Yes
complementaryFilter	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer(optional) 	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Non-Kalman filter based approach: <ul style="list-style-type: none"> Use high and low pass filters to reduce noise in various sensor readings. Fuse the filtered sensor readings based on their assigned weights. 	No

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
insfilterMARG	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer GPS 	Orientation, position, velocity, delta angle bias, delta velocity bias, geomagnetic field vector, magnetometer bias	<p>The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes:</p> <ul style="list-style-type: none"> Piecewise constant linear acceleration. Piecewise constant angular velocity. Accelerometer and gyroscope run at the same rate with no sample dropping. 	Discrete extended Kalman filter	Yes
insfilterAsync	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer GPS 	Orientation, angular velocity, position, velocity, acceleration, accelerometer bias, gyroscope bias, geomagnetic field vector, magnetometer bias	<p>The filter assumes:</p> <ul style="list-style-type: none"> Constant angular velocity Constant acceleration <p>The filter does not require the sensors to be synchronous and each sensor can have sample dropping.</p>	Continuous discrete extended Kalman Filter	Yes

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
insfilterNonholonomic	<ul style="list-style-type: none"> Accelerometer Gyroscope GPS 	Orientation, position, velocity, gyroscope bias, accelerometer bias	<p>The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes:</p> <ul style="list-style-type: none"> Piece-wise constant linear acceleration. Piece-wise constant angular velocity. Accelerometer and gyroscope run at the same rate with no sample dropping. <p>Also, the filter assumes the platform moves forward without side slip.</p>	Discrete extended Kalman Filter	Yes

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
insfilterErrorState	<ul style="list-style-type: none"> • Accelerometer • Gyroscope • Magnetometer • GPS • Visual odometry scale 	Orientation, position, velocity, gyroscope bias, accelerometer bias, and visual odometry scale	<p>The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes:</p> <ul style="list-style-type: none"> • Piece-wise constant linear acceleration. • Piece-wise constant angular velocity. • Accelerometer and gyroscope run at the same rate with no sample dropping. 	Error-state Kalman filter	Yes

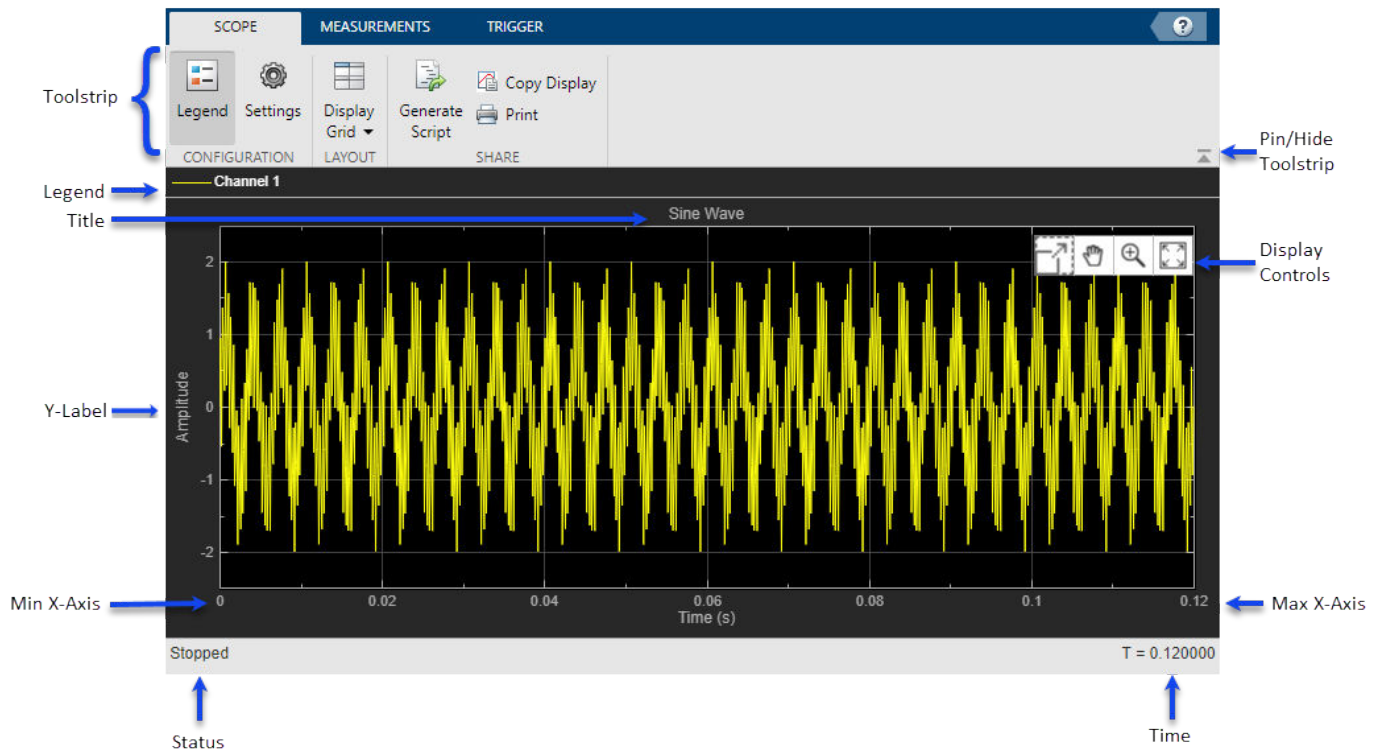
See Also
tunerconfig


Configure Time Scope MATLAB Object

When you use the `timescope` object in MATLAB®, you can configure many settings and tools from the window. These sections show you how to use the Time Scope interface and the available tools.

Signal Display

This figure highlights the important aspects of the Time Scope window in MATLAB.





- **Min X-Axis** — Time scope sets the minimum x -axis limit using the value of the `TimeDisplayOffset` property. To change the **Time Offset** from the Time Scope window, click **Settings** () on the **Scope** tab. Under **Data and Axes**, set the **Time Offset**.
- **Max X-Axis** — Time scope sets the maximum x -axis limit by summing the value of the **Time Offset** property with the span of the x -axis values. If **Time Span** is set to **Auto**, the span of x -axis is $10/\text{SampleRate}$.

The values on the x -axis of the scope display remain the same throughout the simulation.

- **Status** — Provides the current status of the plot. The status can be:
 - **Processing** — Occurs after you run the object and before you run the `release` function.
 - **Stopped** — Occurs after you create the scope object and before you first call the object. This status also occurs after you call `release`.
- **Title, YLabel** — You can customize the title and the y -axis label from **Settings** or by using the `Title` and `YLabel` properties.


- **Toolstrip**

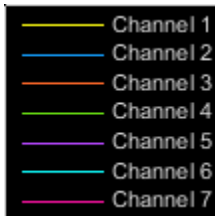
- **Scope** tab — Customize and share the time scope. For example, showing and hiding the legend
- **Measurements** tab — Turn on and control different measurement tools.
- **Trigger** tab — Turn on and modify triggers.

Use the pin button  to keep the toolstrip showing or the arrow button  to hide the toolstrip.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, the legend for a two-channel signal will display the default names

Channel 1, Channel 2. To show the legend, on the **Scope** tab, click **Settings** (). Under **Display and Labels**, select **Show Legend**. If there are a total of seven input channels, the legend displayed is:



By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink® Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the legend. If there are more than seven channels, then the scope repeats this order to assign line colors to the remaining channels. When the axes background is not black, the signals are colored in this order:



To choose line colors or background colors, on the **Scope** tab click **Settings**. Use the **Axes** color pallet to change the background of the plot. Click **Line** to choose a line to change, and the **Color** drop-down to change the line color of the selected line.

Configure Scope Settings

On the **Scope** tab, the **Configuration** section allows you to modify the scope.

- The **Legend** button turns the legend on or off. When you show the legend, you can control which signals are shown. If you click a signal name in the legend, the signal is hidden from the plot and shown in grey on the legend. To redisplay the signal, click on the signal name again. This button corresponds to the ShowLegend property in the object.

- The **Settings** button opens the settings window which allows you to customize the data, axes, display settings, labels, and color settings.

On the **Scope** tab, the **Layout** section allows you to modify the scope layout dimensions.

The **Display Grid** button enables you to select the display layout of the scope.


Use timescope Measurements and Triggers

All measurements are made for a specified channel. By default, measurements are applied to the first channel. To change which channel is being measured, use the **Select Channel** drop-down on the **Measurements** tab.

Data Cursors

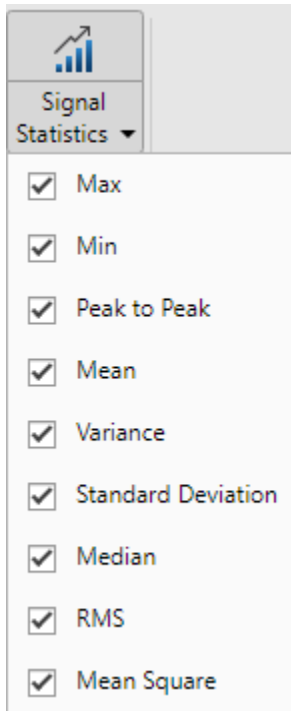
Use the **Data Cursors** button to display screen cursors. Each cursor tracks a vertical line along the signal. The difference between x - and y -values of the signal at the two cursors is displayed in the box between the cursors.

Signal Statistics

Use the **Signal Statistics** button to display statistics about the selected signal at the bottom of the time scope window. You can hide or show the **Statistics** panel using the arrow button  at the bottom right of the panel.

- **Max** — Maximum value within the displayed portion of the input signal.
- **Min** — Minimum value within the displayed portion of the input signal.
- **Peak to Peak** — Difference between the maximum and minimum values within the displayed portion of the input signal.
- **Mean** — Average or mean of all the values within the displayed portion of the input signal.
- **Variance** -- Variance of the values within the displayed portion of the input signal.
- **Standard Deviation** -- Standard deviation of the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean square of the input signal.
- **Mean Square** -- Mean square of the values within the displayed portion of the input signal.

To customize which statistics to show and compute, use the **Signal Statistics** list.



Peak Finder

Use the **Peak Finder** button to display peak values for the selected signal. Peaks are defined as a local maximum where lower values are present on both sides of a peak. End points are not considered peaks. For more information on the algorithms used, see the `findpeaks` function.

When you turn on the peak finder measurements, an arrow appears on the plot at each maxima and a Peaks panel appears at the bottom of the timescope window showing the x and y values at each peak.

You can customize several peak finder settings:

- **Num Peaks** — The number of peaks to show. Must be a scalar integer from 1 through 99.
- **Min Height** — The minimum height difference between a peak and its neighboring samples.
- **Min Distance** — The minimum number of samples between adjacent peaks.
- **Threshold** — The level above which peaks are detected.
- **Label Peaks** — Show labels (**P1**, **P2**, ...) above the arrows on the plot.

Bilevel Measurements

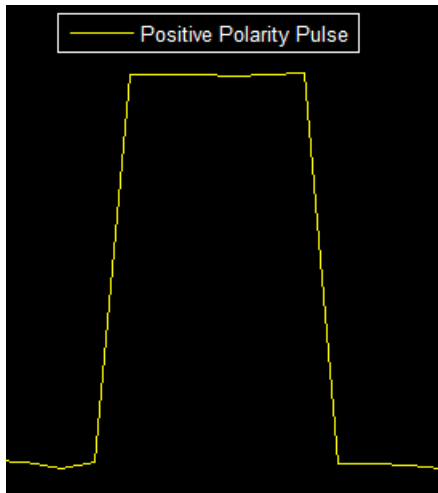
With bilevel measurements, you can measure transitions, aberrations, and cycles.

Bilevel Settings

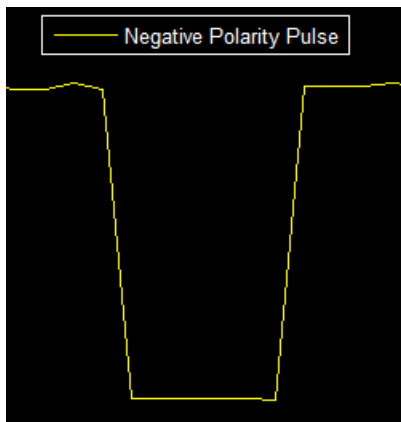
When using bilevel measurements, you can set these properties:

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel detects the high- and low-state levels of a bilevel waveform. When this check box is cleared, you can enter in values for the high- and low-state levels manually.

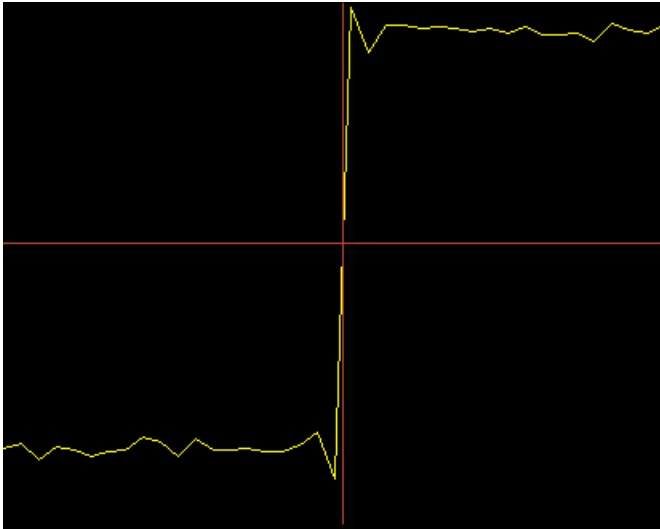
- **High** — Manually specify the value that denotes a positive polarity or high-state level.



- **Low** — Manually specify the value that denotes a negative polarity or low-state level.



- **State Level Tol. %** — Tolerance levels within which the initial and final levels of each transition must lie to be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low-state levels. In the following figure, the mid-reference level is shown as a horizontal line, and its corresponding mid-reference level instant is shown as a vertical line.

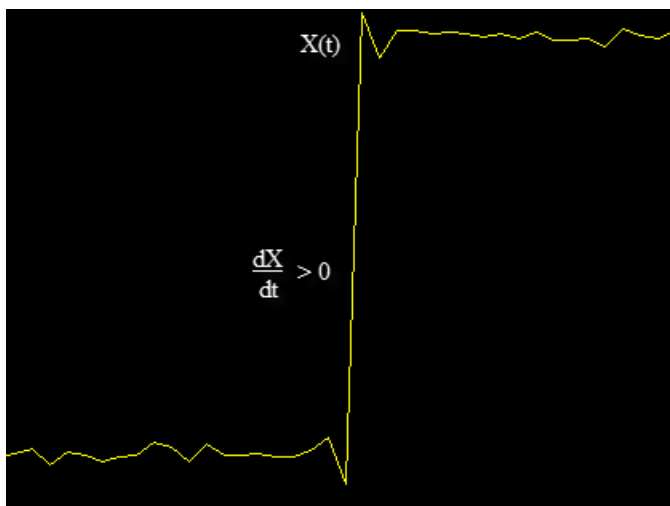


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs is used for computing a valid settling time. Settling time is displayed in the **Aberrations** pane.

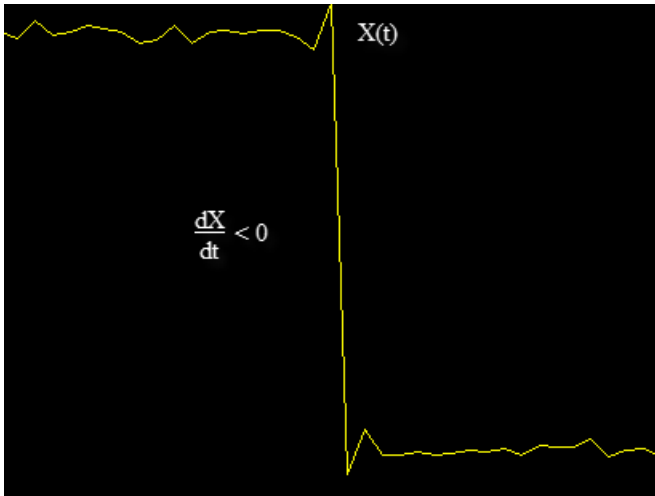
Transitions

Select **Transitions** to display calculated measurements associated with the input signal changing between its two possible state level values, high and low. The measurements are displayed in the **Transitions** pane at the bottom of the scope window.

The **+ Edges** row measures rising edges or a positive-going transition. A rising edge in a bilevel waveform is a transition from the low-state level to the high-state level with a slope value greater than zero.



The **- Edges** row measures falling edges or a negative-going transition. A falling edge in a bilevel waveform is a transition from the high-state level to the low-state level with a slope value less than zero.

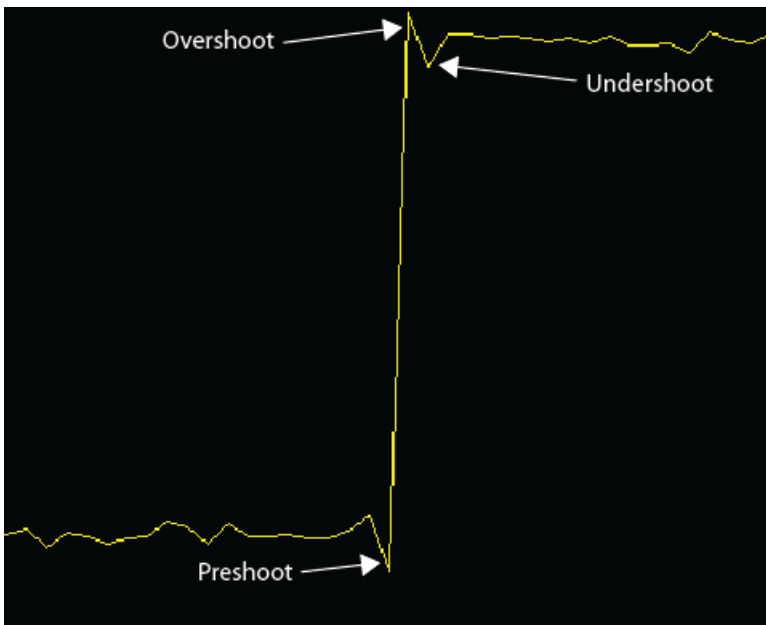


The Transition measurements assume that the amplitude of the input signal is in units of volts. For the transition measurements to be valid, you must convert all input signals to volts.

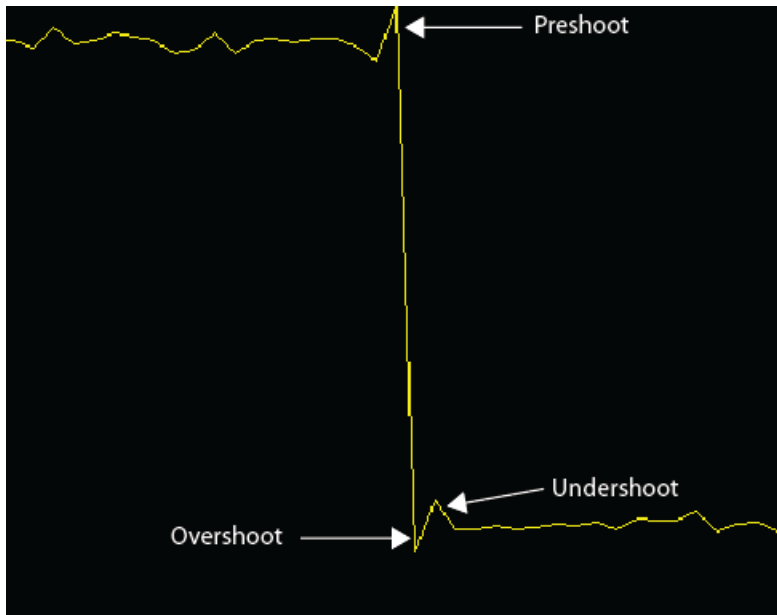
Aberrations

Select **Aberrations** to display calculated measurements involving the distortion and damping of the input signal such as preshoot, overshoot, and undershoot. Overshoot and undershoot, respectively, refer to the amount that a signal exceeds and falls below its final steady-state value. Preshoot refers to the amount before a transition that a signal varies from its initial steady-state value. The measurements are displayed in the **Transitions** pane at the bottom of the scope window.

This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.



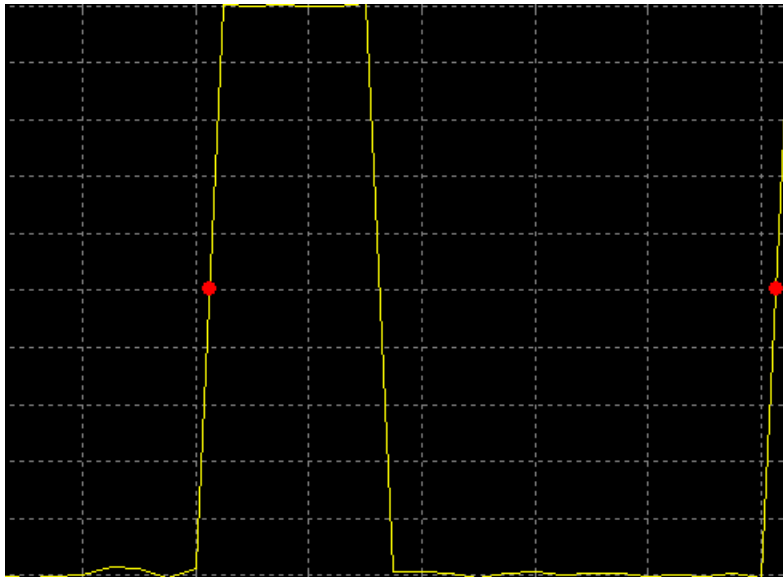
The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.



Cycles

Select **Cycles** calculates repetitions or trends in the displayed portion of the input signal. The measurements are displayed in the **Cycles** pane at the bottom of the scope window in two rows: **+ Pulses** for the positive-polarity pulses and **- Pulses** for the negative-polarity pulses.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. To calculate period, the timescope takes the difference between the mid-reference level instants of the initial transition of each pulse and the next identical-polarity transition. These mid-reference level instants for a positive-polarity pulse appear as red dots in the following figure.



- **Frequency** — Reciprocal of the average period, measured in hertz.

- **Count** — Number of positive- or negative-polarity pulses counted.
- **Width** — Average duration between rising and falling edges of each pulse within the displayed portion of the input signal.
- **Duty Cycle** — Average ratio of pulse width to pulse period for each pulse within the displayed portion of the input signal.

Triggers

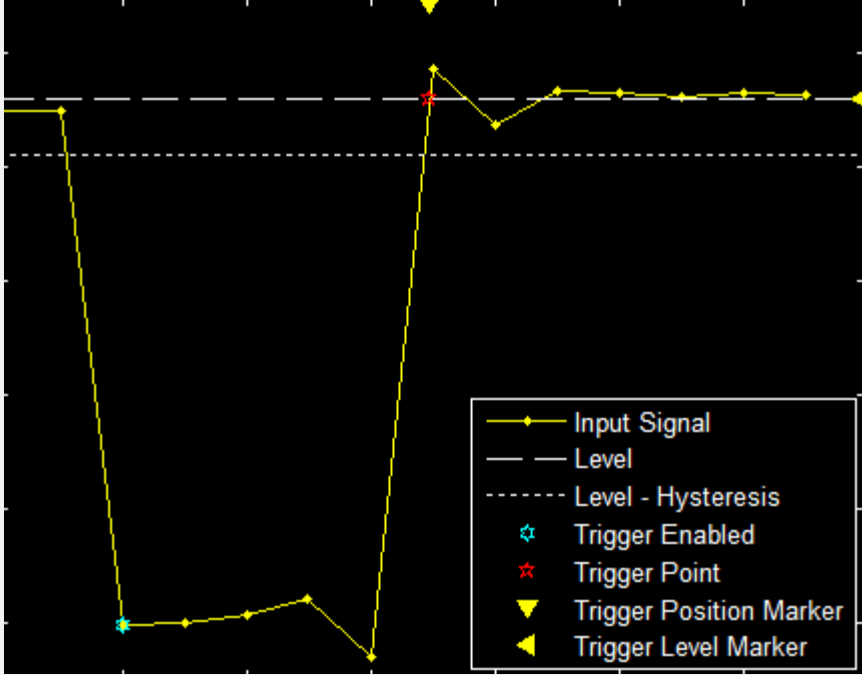
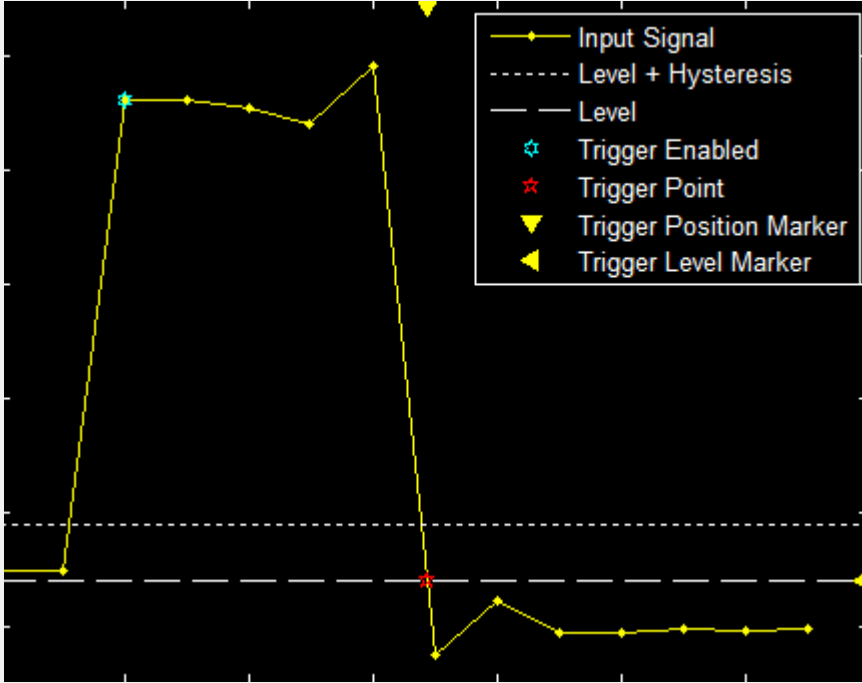
Define a trigger event to identify the simulation time of specified input signal characteristics. You can use trigger events to stabilize periodic signals such as a sine wave or capture non-periodic signals such as a pulse that occurs intermittently.

To define a trigger:

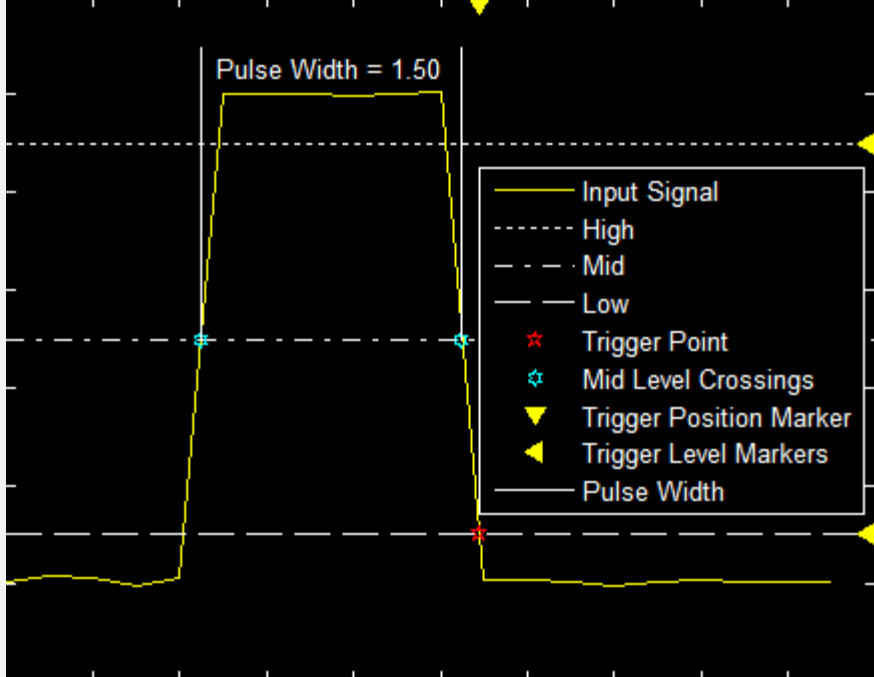
- 1 On the Trigger tab of the scope window, select the channel you want to trigger.
- 2 Specify when the display updates by selecting a triggering **Mode**.
 - **Auto** — Display data from the last trigger event. If no event occurs after one time span, display the last available data.
Normal — Display data from the last trigger event. If no event occurs, the display remains blank.
 - **Once** — Display data from the last trigger event and freeze the display. If no event occurs, the display remains blank. Click the **Rearm** button to look for the next trigger event.
- 3 Select a triggering type, polarity, and any other properties. See the Trigger Properties table.
- 4 Click **Enable Trigger**.

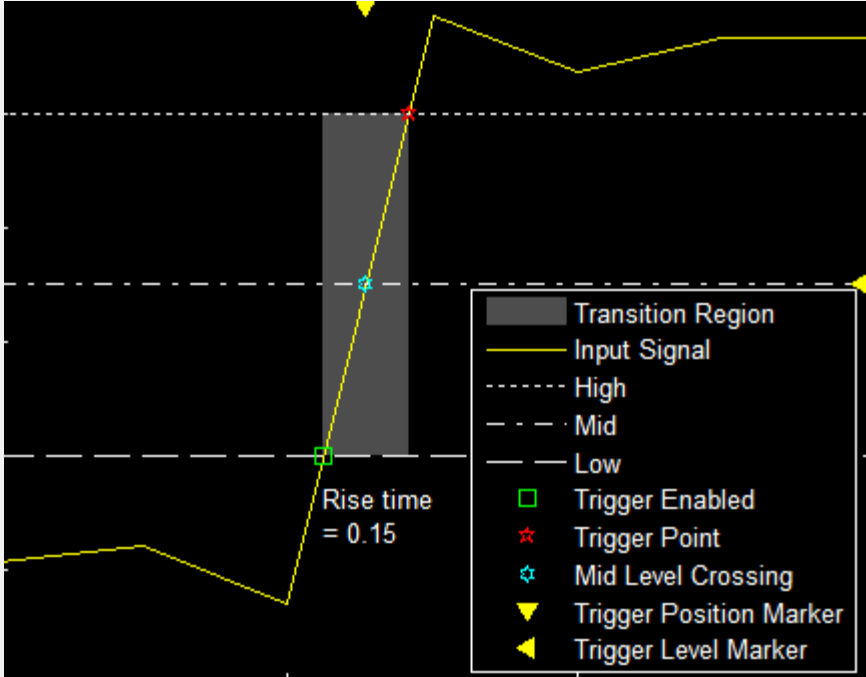
You can set the trigger position to specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.

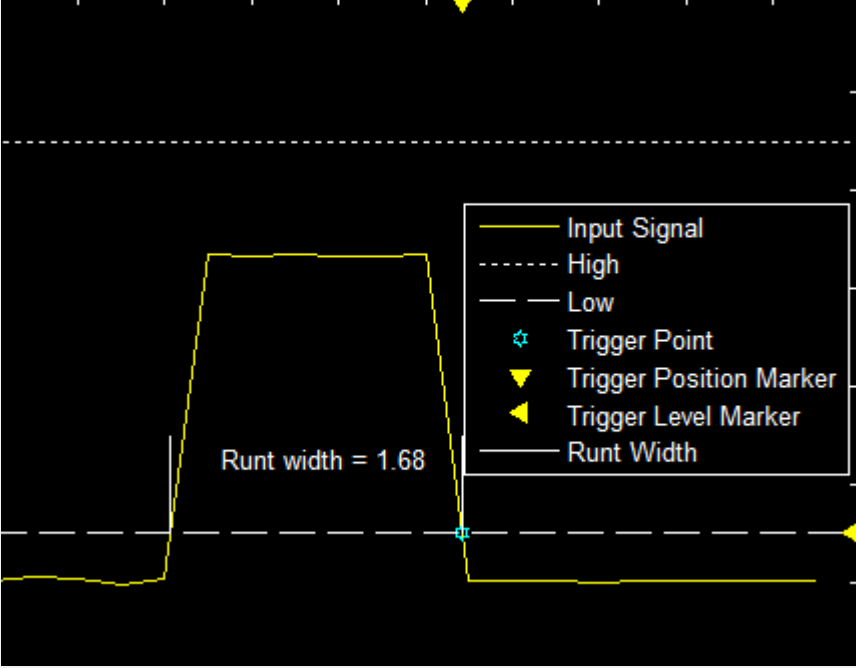
Trigger Properties


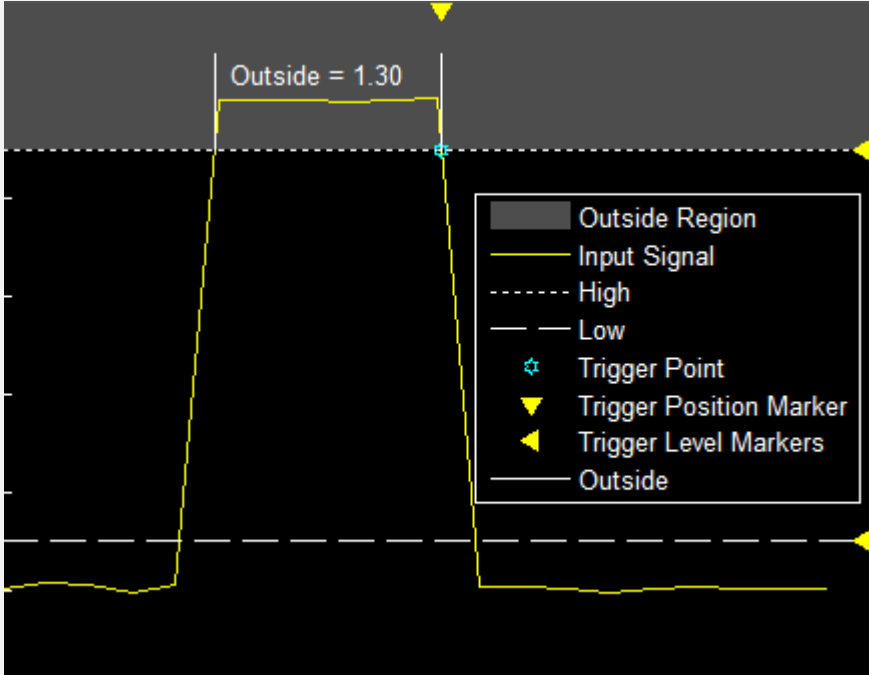
Trigger Type	Trigger Parameters
Edge — Trigger when the signal crosses a threshold.	<p>Polarity — Select the polarity for an edge-triggered signal.</p> <ul style="list-style-type: none"> • Rising — Trigger when the signal is increasing.  <ul style="list-style-type: none"> • Falling — Trigger when the signal value is decreasing. 

Trigger Type	Trigger Parameters
	<ul style="list-style-type: none">• Either — Trigger when the signal is increasing or decreasing.Level — Enter a threshold value for an edge-triggered signal. Auto level is 50%Hysteresis — Enter a value for an edge-triggered signal. See “Hysteresis of Trigger Signals” on page 2-30Delay — Offset the trigger by a fixed delay in seconds.Holdoff — Set the minimum possible time between triggers.Position — Set horizontal position of the trigger on the screen.

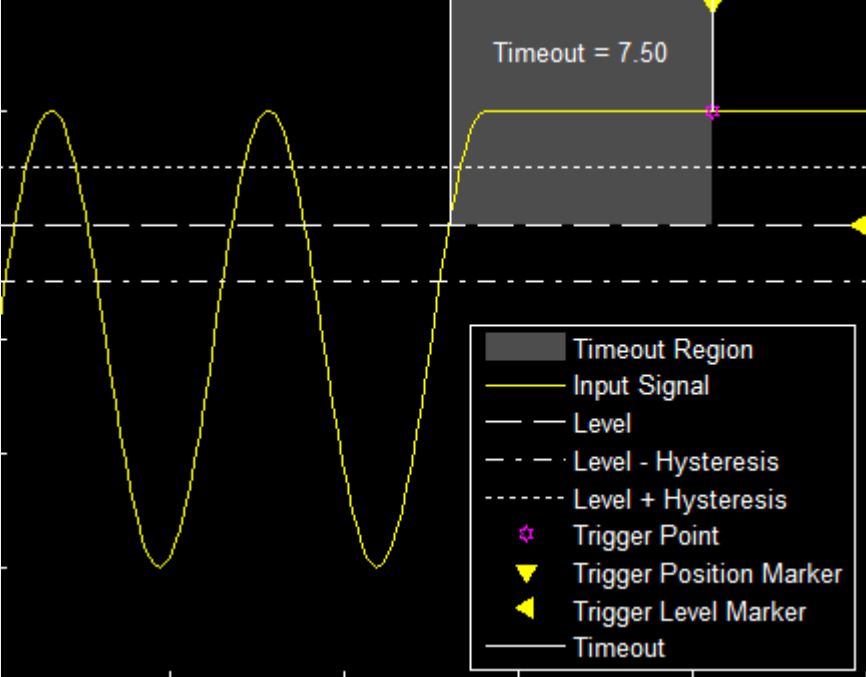
Trigger Type	Trigger Parameters
<p>Pulse Width — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.</p>	<p>Polarity — Select the polarity for a pulse width-triggered signal.</p> <ul style="list-style-type: none"> • Positive — Trigger on a positive-polarity pulse when the pulse crosses the low threshold for a second time.  <ul style="list-style-type: none"> • Negative — Trigger on a negative-polarity pulse when the pulse crosses the high threshold for a second time. • Either — Trigger on both positive-polarity and negative-polarity pulses.
	<p>Note A glitch-trigger is a special type of a pulse width-trigger. A glitch-trigger occurs for a pulse or spike whose duration is less than a specified amount. You can implement a glitch-trigger by using a pulse-width-trigger and setting the Max Width parameter to a small value.</p>
	<p>High — Enter a high value for a pulse-width-triggered signal. Auto level is 90%.</p>
	<p>Low — Enter a low value for a pulse-width-triggered signal. Auto level is 10%.</p>
	<p>Min Width — Enter the minimum pulse width for a pulse-width-triggered signal. Pulse width is measured between the first and second crossings of the middle threshold.</p>
	<p>Max Width — Enter the maximum pulse width for a pulse-width-triggered signal.</p>
	<p>Delay — Offset the trigger by a fixed delay in seconds.</p>

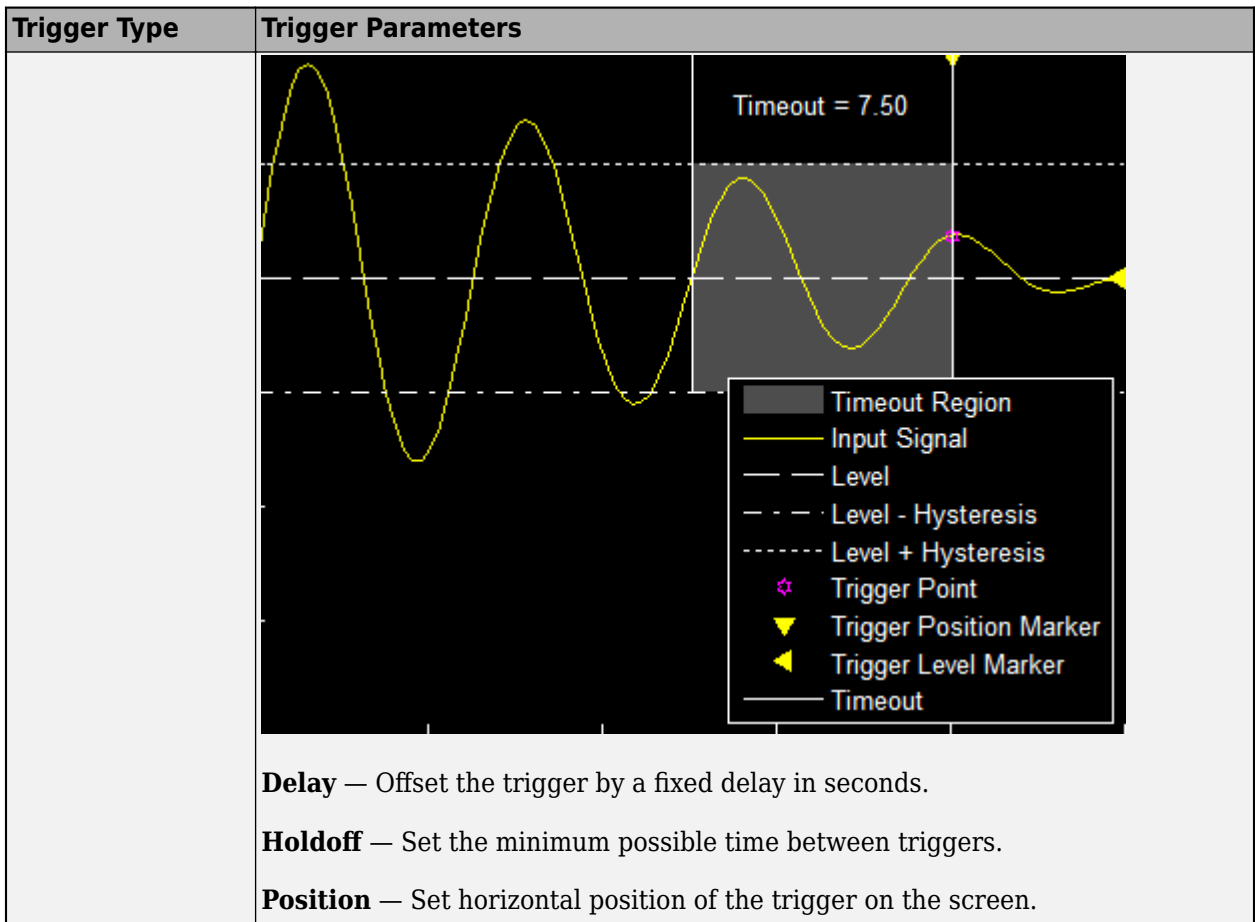
Trigger Type	Trigger Parameters
	<p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>
<p>Transition — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.</p>	<p>Polarity — Select the polarity for a transition-triggered signal.</p> <ul style="list-style-type: none"> • Rise Time — Trigger on an increasing signal when the signal crosses the high threshold.  <ul style="list-style-type: none"> • Fall Time — Trigger on a decreasing signal when the signal crosses the low threshold. • Either — Trigger on an increasing or decreasing signal. <p>High — Enter a high value for a transition-triggered signal. Auto level is 90%.</p> <p>Low — Enter a low value for a transition-triggered signal. Auto level is 10%.</p> <p>Min Time — Enter a minimum time duration for a transition-triggered signal.</p> <p>Max Time — Enter a maximum time duration for a transition-triggered signal.</p> <p>Delay — Offset the trigger by a fixed delay in seconds.</p> <p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>

Trigger Type	Trigger Parameters
<p>Runt— Trigger when a signal crosses a low threshold or a high threshold twice within a specified time.</p>	<p>Polarity — Select the polarity for a runt-triggered signal.</p> <ul style="list-style-type: none"> • Positive — Trigger on a positive-polarity pulse when the signal crosses the low threshold a second time without crossing the high threshold.  <ul style="list-style-type: none"> • Negative — Trigger on a negative-polarity pulse. • Either — Trigger on both positive-polarity and negative-polarity pulses. <p>High — Enter a high value for a runt-triggered signal. Auto level is 90%.</p> <p>Low — Enter a low value for a runt-triggered signal. Auto level is 10%.</p> <p>Min Width — Enter a minimum width for a runt-triggered signal. Pulse width is measured between the first and second crossing of a threshold.</p> <p>Max Width — Enter a maximum pulse width for a runt-triggered signal.</p> <p>Delay — Offset the trigger by a fixed delay in seconds.</p> <p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>

Trigger Type	Trigger Parameters
<p>Window — Trigger when a signal stays within or outside a region defined by the high and low thresholds for a specified time.</p>	<p>Polarity — Select the region for a window-triggered signal.</p> <ul style="list-style-type: none"> • Inside — Trigger when a signal leaves a region between the low and high levels.  <ul style="list-style-type: none"> • Outside — Trigger when a signal enters a region between the low and high levels. 

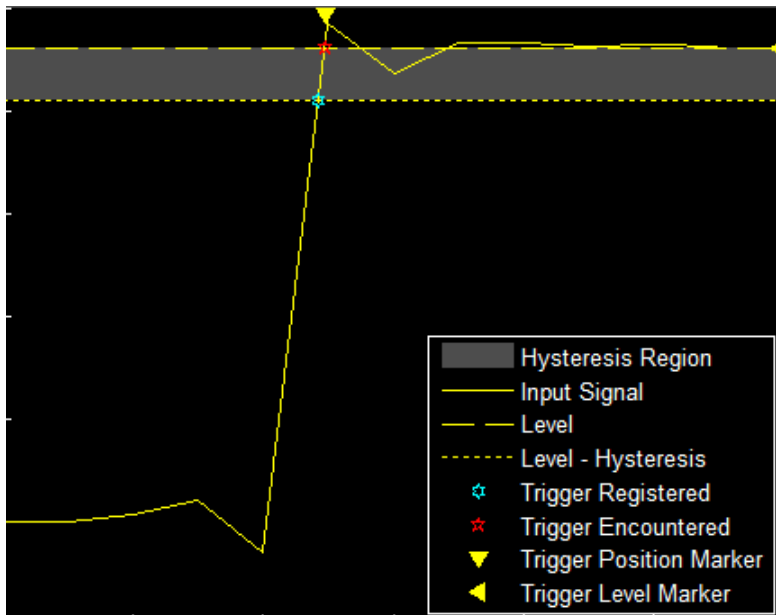
Trigger Type	Trigger Parameters
	<ul style="list-style-type: none"><li data-bbox="493 296 1451 352">• Either — Trigger when a signal leaves or enters a region between the low and high levels. <p data-bbox="493 380 1430 411">High — Enter a high value for a window-triggered signal. Auto level is 90%.</p> <p data-bbox="493 438 1377 470">Low — Enter a low value for a window-trigger signal. Auto level is 10%.</p> <p data-bbox="493 497 1442 529">Min Time — Enter the minimum time duration for a window-triggered signal.</p> <p data-bbox="493 556 1451 588">Max Time — Enter the maximum time duration for a window-triggered signal.</p> <p data-bbox="493 615 1154 646">Delay — Offset the trigger by a fixed delay in seconds.</p> <p data-bbox="493 674 1227 705">Holdoff — Set the minimum possible time between triggers.</p> <p data-bbox="493 732 1263 764">Position — Set horizontal position of the trigger on the screen.</p>

Trigger Type	Trigger Parameters
<p>Timeout — Trigger when a signal stays above or below a threshold longer than a specified time</p>	<p>Polarity — Select the polarity for a timeout-triggered signal.</p> <ul style="list-style-type: none"> • Rising — Trigger when the signal does not cross the threshold from below. For example, if you set Timeout to 7.50 seconds, the scope triggers 7.50 seconds after the signal crosses the threshold.  <ul style="list-style-type: none"> • Falling — Trigger when the signal does not cross the threshold from above. • Either — Trigger when the signal does not cross the threshold from either direction <p>Level — Enter a threshold value for a timeout-triggered signal.</p> <p>Hysteresis — Enter a value for a timeout-triggered signal. See “Hysteresis of Trigger Signals” on page 2-30.</p> <p>Timeout — Enter a time duration for a timeout-triggered signal.</p> <p>Alternatively, a trigger event can occur when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the threshold.</p>

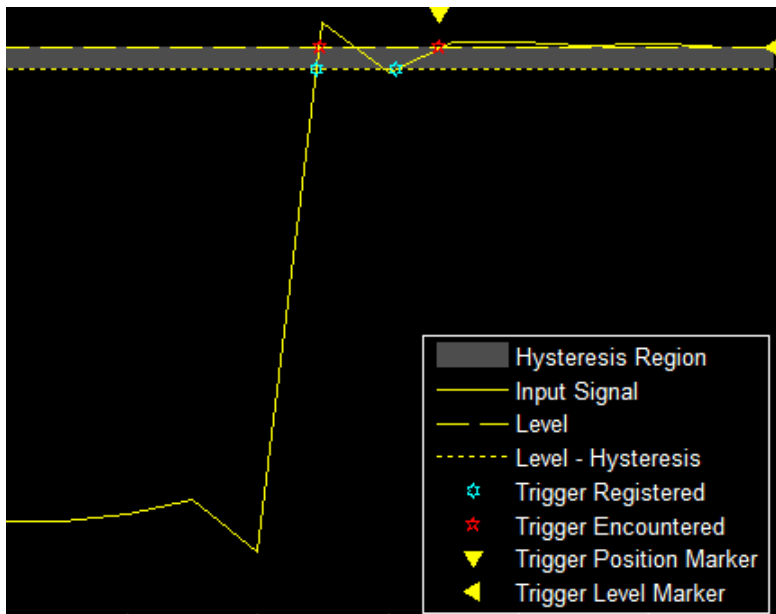


Hysteresis of Trigger Signals

Hysteresis — Specify the hysteresis or noise-reject value. This parameter is visible when you set **Type** to Edge or Timeout. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. In the case of an edge trigger with rising polarity, the scope ignores the times that a signal crosses the trigger level within the hysteresis region.



You can reduce the hysteresis region size by decreasing the hysteresis value. In this example, if you set the hysteresis value to 0.07, the scope also considers the second rising edge to be a trigger event.



Share or Save the Time Scope





If you want to save the time scope for future use or share it with others, use the buttons in the **Share** section of the **Scope** tab.

- **Generate Script** — Generate a script to re-create your time scope with the same settings. An editor window opens with the code required to re-create your `timescope` object.

- **Copy Display** — Copy the display to your clipboard. You can paste the image in another program to save or share it.
- **Print** — Opens a print dialog box from which you can print out the plot image.

Scale Axes

To scale the plot axes, you can use the mouse to pan around the axes and the scroll button on your mouse to zoom in and out of the plot. Additionally, you can use the buttons that appear when you hover over the plot window.

-  — Maximize the axes, hiding all labels and inseting the axes values.
-  — Zoom in on the plot.
-  — Pan the plot.
-  — Autoscale the axes to fit the shown data.

See Also

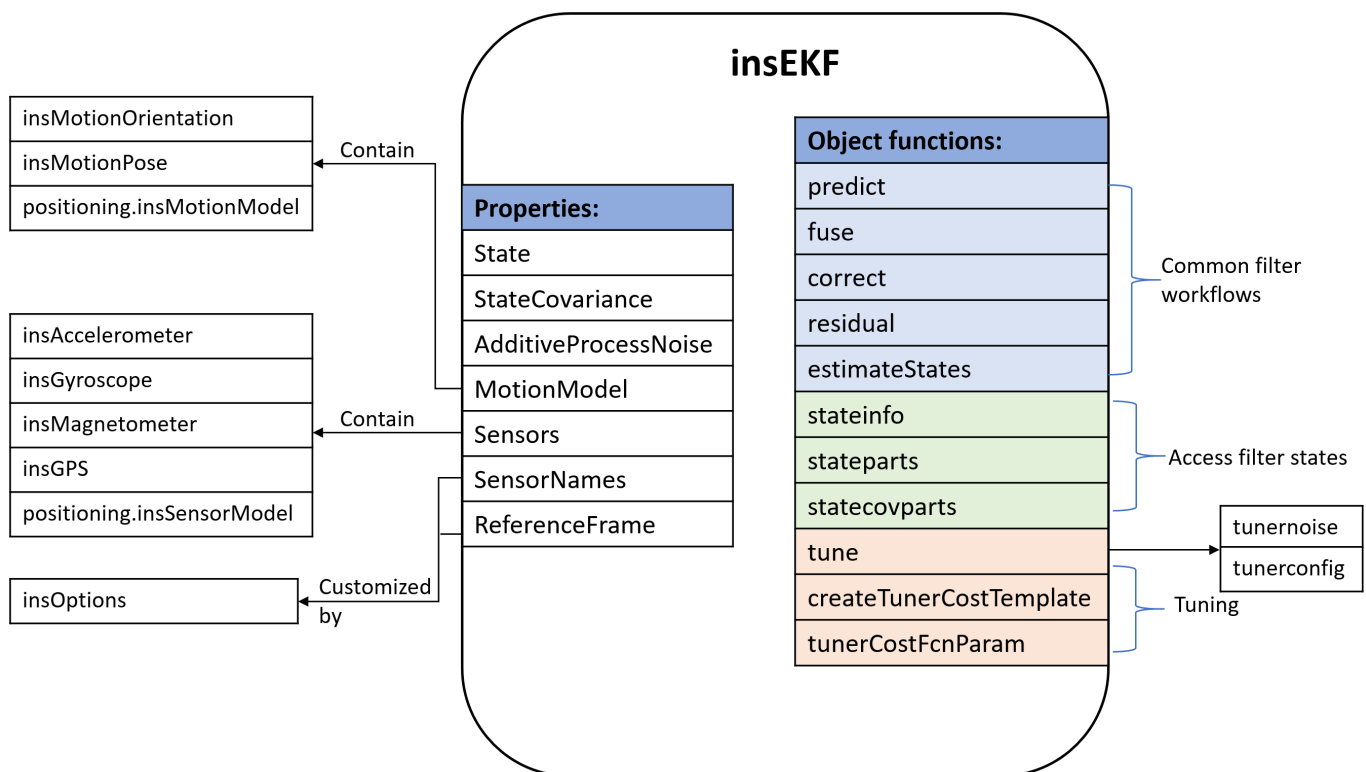
Objects

timescope

Fuse Inertial Sensor Data Using insEKF-Based Flexible Fusion Framework

The `insEKF` filter object provides a flexible framework that you can use to fuse inertial sensor data. You can fuse measurement data from various inertial sensors by selecting or customizing the sensor models used in the filter, and estimate different platform states by selecting or customizing the motion model used in the filter. The `insEKF` object is based on a continuous-discrete extended Kalman filter, in which the state prediction step is continuous, and the measurement correction or fusion step is discrete.

Overall, the object contains 7 properties and 11 object functions. The object also interacts with other objects and functions to support regular filter workflows, as well as filter tuning.



Object Properties

The `insEKF` object maintains major filter variables and information of the motion model and sensor models using these properties:

- **State** — Platform state saved in the filter, specified as a vector. This property maintains the whole state vector, including the platform states defined by the motion model, such as orientation, position, and velocity, as well as the sensor states defined by sensor models, such as sensor biases.

You can interact with this property in various ways:

- Display the state information saved in the vector by using the `stateinfo` object function.

- Directly specify this property. This may not be convenient since the dimension of the state vector can be large.
- Get or set a part or component of the state vector by using the `stateparts` object function.
- **StateCovariance** — State estimate error covariance, specified as a matrix. This property maintains the whole covariance matrix corresponding to the state vector. You can interact with this property in various ways:
 - Directly specify this property. This may not be convenient since the dimension of the covariance matrix can be large.
 - Get or set a part of the covariance matrix by using the `statecovparts` object function.
- **AdditiveProcessNoise** — Additive process noise covariance, specified as a matrix. This property maintains the whole additive process noise covariance matrix corresponding to the state vector.
- **MotionModel** — Motion model to predict the filter state, specified as an object. You can use one of the these options to specify this property when constructing the filter:
 - `insMotionOrientation` — Model orientation-only platform motion assuming a *constant angular velocity*. Using this object adds the quaternion and angular velocity state to the `State` property.
 - `insMotionPose` — Model 3-D motion assuming *constant angular velocity* and *constant linear acceleration*. Using this model adds the quaternion, angular velocity, position, linear velocity, and acceleration state to the `State` property.
 - `positioning.INSMotionModel` — An interface class to implement a motion model object used with the filter. To customize a motion model object, you must inherit from this interface class and implement at least two methods: `modelstates` and `stateTransition`.
- **Sensors** — Sensor models that model the corresponding sensor measurement based on the platform state, specified as a cell array of INS sensor objects. You can specify each INS sensor object using one of the these options:
 - `insAccelerometer` — Model accelerometer readings, including the gravitational acceleration and sensor bias by default. The model also includes the sensor acceleration if the `State` property of the filter includes the `Acceleration` state.
 - `insMagnetometer` — Model magnetometer readings, including the geomagnetic vector and the sensor bias.
 - `insGyroscope` — Model gyroscope readings, including the angular velocity vector and the sensor bias.
 - `insGPS` — Model GPS readings, including the latitude, longitude, and altitude coordinates. If you fuse velocity data from the GPS sensor, the object additionally models velocity measurements.
 - `positioning.INSSensorModel` — An interface class to implement a sensor model object used with the filter. To customize a sensor model object, you must inherit from this interface class and implement at least the `measurement` method.
- **SensorNames** — Names of the sensors added to the filter, specified as a cell array of character vector. The filter object assigns default names to the added sensors. These sensor names are useful when you use various object functions of the filter.
 - To customize the sensor names, you must use the `insOptions` object.

- `ReferenceFrame` — Reference frame of the extended Kalman filter object, specified as "NED" for the north-east-down frame by default. To customize it as "ENU" for the east-north-up frame, you must use the `insOptions` object.

Note You can also specify the data type used in the filter as `double` (default) or `single` by specifying the `DataType` property of the `insOptions` object.

Object Functions

The `insEKF` object provides various object functions for implementing common filter workflows, accessing filter states and covariances, and filter tuning.

These object functions support common filter workflows:

- `predict` — Predict the filter state forward in time based on the motion model used in the filter.
- `fuse` — Fuse or correct the filter state using a measurement based on a sensor model previously added to the filter. You can use this object function multiple times if you need to fuse multiple measurements.
- `correct` — Correct the filter using direct state measurement of the filter state. A direct measurement contains a subset of the filter state vector elements. You can use this object function multiple times for multiple direct measurements.
- `residual` — Return the residual and residual covariance of a measurement based on the current state of the filter.
- `estimateStates` — Obtain the state estimates based on a timetable of sensor data. The object function processes the measurements one-by-one and returns the corresponding state estimates.

These object functions enable you to access various states and variables maintained by the filter:

- `stateinfo` — Return or display the state components saved in the `State` property of the filter. Using this function, you can observe what components the state consists of and the indices for all the state components.
- `stateparts` — Get or set parts of the state vector. Since the state vector contain many components, this object function enables you to get or set only a component of the whole state vector.
- `statecovparts` — Get or set parts of the state estimate error covariance matrix. Similar to the `stateparts` function, this object function enables you to get or set only a part of the state estimate error covariance matrix.

To obtain more accurate state estimates, you often need to tune the filter parameters to reduce estimate errors. The `insEKF` object provides these object functions to facilitate filter tuning:

- `tune` — Adjust the `AdditiveProcessNoise` property of the filter object and measurement noise of the sensors to reduce the state estimation error between the filter estimates and the ground truth.
 - You can use the `tunernoise` function to obtain an example for the measure noise structure, required by the `tune` function.
 - You can optionally use the `tunerconfig` object to specify tuning parameters, such as function tolerance and the cost function, as well as which elements of the process noise matrix to tuned.

- `createTunerCostTemplate` — Creates a template for a tuner cost function that you can further modify to customize your own cost function.
- `tunerCostFcnParam` — Creates a required structure for tuning an `insEKF` filter with a custom cost function. The structure is useful when generating C-code for a cost function using MATLAB Coder™.

Example: Fuse Inertial Sensor Data Using `insEKF`

This example introduces the basic workflows for fusing inertial sensor data using the `insEKF` object, which supports a flexible fusion framework based on a continuous discrete Kalman filter.

Create `insEKF` Object

Create an `insEKF` object directly.

```
filter1 = insEKF

filter1 =
  insEKF with properties:
        State: [13×1 double]
   StateCovariance: [13×13 double]
 AdditiveProcessNoise: [13×13 double]
      MotionModel: [1×1 insMotionOrientation]
         Sensors: {[1×1 insAccelerometer] [1×1 insGyroscope]}
   SensorNames: {'Accelerometer' 'Gyroscope'}
 ReferenceFrame: 'NED'
```

From the `Sensors` property, note that the object contains two sensor models, `insAccelerometer` and `insGyroscope` by default. These enables you to fuse accelerometer and gyroscope data, respectively. From the `MotionModel` property, note that the object defaults to an `insMotionOrientation` model, which models rotation-only motion and not translational motion. Due to the specified motion model and sensor models, the state of the filter is a 13-by-1 vector. Get the components and corresponding indices from the state vector using the `stateinfo` object function.

```
stateinfo(filter1)

ans = struct with fields:
   Orientation: [1 2 3 4]
 AngularVelocity: [5 6 7]
 Accelerometer_Bias: [8 9 10]
 Gyroscope_Bias: [11 12 13]
```

Note that, in addition to the orientation and angular velocity states, the filter also includes the accelerometer bias and gyroscope bias.

You can explicitly specify the motion model and sensor models when constructing the filter. For example, create an `insEKF` object and specify the motion model as an `insMotionPose` object, which models both rotational motion and translational motion, and specify the sensors as an `insAccelerometer`, an `insGPS`, and another `insGPS` object. This enables the fusion of one set of accelerometer data and two sets of GPS data.

```
filter2 = insEKF(insAccelerometer,insGPS,insGPS,insMotionPose)
```

```

filter2 =
  insEKF with properties:
      State: [19x1 double]
      StateCovariance: [19x19 double]
      AdditiveProcessNoise: [19x19 double]
      MotionModel: [1x1 insMotionPose]
      Sensors: {[1x1 insAccelerometer] [1x1 insGPS] [1x1 insGPS]}
      SensorNames: {'Accelerometer' 'GPS' 'GPS_1'}
      ReferenceFrame: 'NED'

```

The `State` property is a 19-by-1 vector that contains these components:

```

stateinfo(filter2)

ans = struct with fields:
    Orientation: [1 2 3 4]
    AngularVelocity: [5 6 7]
    Position: [8 9 10]
    Velocity: [11 12 13]
    Acceleration: [14 15 16]
    Accelerometer_Bias: [17 18 19]

```

The `SensorNames` property enables you to indicate specific sensors when using various object functions of the filter. The filter generates the names for added sensors in a default format. To provide custom names for the sensors, you must use the `insOptions` object. The `insOptions` object can also specify the data type of variables used in the filter and the `ReferenceFrame` of the filter.

```

options = insOptions(SensorNamesSource="Property", ...
    SensorNames={'Sensor1','Sensor2','Sensor3'}, ...
    Datatype="single", ...
    ReferenceFrame="ENU");
filter3 = insEKF(insAccelerometer,insGPS,insGPS,insMotionPose,options)

filter3 =
  insEKF with properties:
      State: [19x1 single]
      StateCovariance: [19x19 single]
      AdditiveProcessNoise: [19x19 single]
      MotionModel: [1x1 insMotionPose]
      Sensors: {[1x1 insAccelerometer] [1x1 insGPS] [1x1 insGPS]}
      SensorNames: {'Sensor1' 'Sensor2' 'Sensor3'}
      ReferenceFrame: 'ENU'

```

You can also directly obtain the indices of a state component based on the sensor names. For example,

```

stateinfo(filter3,'Sensor1_Bias')

ans = 1x3
     17     18     19

```

Configure Filter Properties

Create a new `insEKF` object with an accelerometer and a gyroscope. Explicitly define these two sensors for later usage.

```
accSensor = insAccelerometer;
gyroSensor = insGyroscope;
filter = insEKF(accSensor,gyroSensor)

filter =
  insEKF with properties:
        State: [13×1 double]
    StateCovariance: [13×13 double]
AdditiveProcessNoise: [13×13 double]
        MotionModel: [1×1 insMotionOrientation]
          Sensors: {[1×1 insAccelerometer] [1×1 insGyroscope]}
    SensorNames: {'Accelerometer' 'Gyroscope'}
    ReferenceFrame: 'NED'
```

Load prerecorded data for an accelerometer and a gyroscope. The sample rate of the recorded data is 100 Hz. It contains the sensor data, ground truth, and the initial orientation represented by a quaternion.

```
ld = load("accelGyroINSEKFData.mat")

ld = struct with fields:
    sampleRate: 100
    sensorData: [300×2 timetable]
    groundTruth: [300×1 timetable]
    initOrient: [1×1 quaternion]
```

Before fusing the sensor data, you need to set up the initial orientation for the filter state. First, observe the state information.

```
stateinfo(filter)

ans = struct with fields:
    Orientation: [1 2 3 4]
    AngularVelocity: [5 6 7]
    Accelerometer_Bias: [8 9 10]
    Gyroscope_Bias: [11 12 13]
```

Query the index of a state component directly by using the corresponding sensor.

```
stateinfo(filter,accSensor,"Bias")

ans = 1×3
     8     9    10
```

Set only the `Orientation` component of the state vector using the `stateparts` object function.


```

quatElements = compact(ld.initOrient); % Convert the quaternion object to a vector of four elements
stateparts(filter,"Orientation",quatElements); % Specify the Orientation state component
stateparts(filter,"Orientation") % Show the specified Orientation state component

```

```
ans = 1x4
```

```
1.0000 -0.0003 0.0001 0.0002
```

Specify the covariance matrix corresponding to the orientation as a diagonal matrix.

```
statecovparts(filter,"Orientation",1e-2);
```

Fuse Sensor Data

You can fuse sensor data by recursively calling the `predict` and `fuse` object functions.

Preallocate variables for saving estimated results.

```

N = size(ld.sensorData,1);
estOrient = quaternion.zeros(N,1);
dt = seconds(diff(ld.sensorData.Properties.RowTimes));

```

Predict the filter state, fuse the sensor data, and obtain the estimates.

```

for ii = 1:N
    if ii ~= 1
        % Predict forward in time.
        predict(filter,dt(ii-1));
    end
    % Fuse accelerometer data.
    fuse(filter,accSensor,ld.sensorData.Accelerometer(ii,:),1);
    % Fuse gyroscope data.
    fuse(filter,gyroSensor,ld.sensorData.Gyroscope(ii,:),1);
    % Extract the orientation state estimate using the stateparts object function.
    estOrient(ii) = quaternion(stateparts(filter,"Orientation"));
end

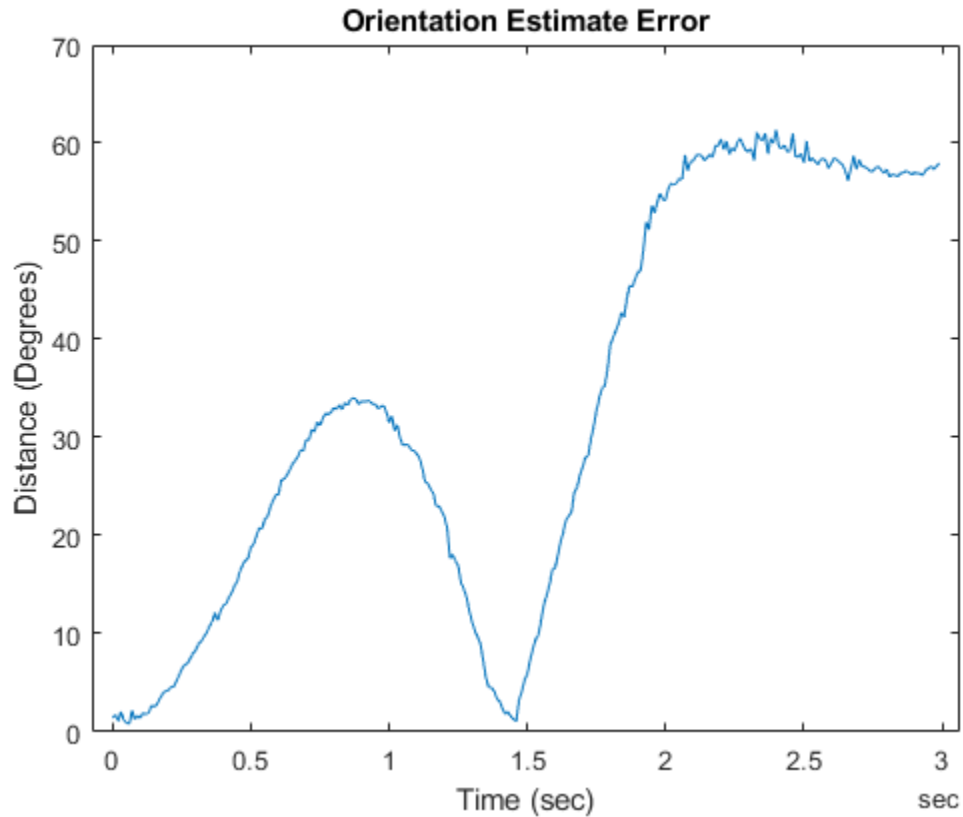
```

Visualize the estimate error, in quaternion distance, using the `dist` object function of the quaternion object.

```

figure
times = ld.groundTruth.Properties.RowTimes;
distance = rad2deg(dist(estOrient,ld.groundTruth.Orientation));
plot(times,distance)
xlabel("Time (sec)")
ylabel("Distance (Degrees)")
title("Orientation Estimate Error")

```



The results indicate that the estimate errors are large. You can also use the `estimateStates` object function to process the sensor data and obtain the same results.

Tune Filter

Given that the estimation results are not ideal, you can try to tune the filter parameters to reduce estimate errors.

Create a measurement noise structure and a `tunerconfig` object used to configure the tuning parameters.

```
mnoise = tunernoise(filter);
cfg = tunerconfig(filter,MaxIterations=10,ObjectiveLimit=1e-4);
```

Reinitialize the filter. Use the `tune` object function to tune the filter and obtain the tuned noise.

```
stateparts(filter,"Orientation",quatElements);
statecovparts(filter,"Orientation",1e-2);

tunedmn = tune(filter,mnoise,ld.sensorData,ld.groundTruth,cfg);
```

Iteration	Parameter	Metric
1	AdditiveProcessNoise(1)	0.3787
1	AdditiveProcessNoise(15)	0.3761
1	AdditiveProcessNoise(29)	0.3695
1	AdditiveProcessNoise(43)	0.3655
1	AdditiveProcessNoise(57)	0.3533

1	AdditiveProcessNoise(71)	0.3446
1	AdditiveProcessNoise(85)	0.3431
1	AdditiveProcessNoise(99)	0.3428
1	AdditiveProcessNoise(113)	0.3427
1	AdditiveProcessNoise(127)	0.3426
1	AdditiveProcessNoise(141)	0.3298
1	AdditiveProcessNoise(155)	0.3206
1	AdditiveProcessNoise(169)	0.3200
1	AccelerometerNoise	0.3199
1	GyroscopeNoise	0.3198
2	AdditiveProcessNoise(1)	0.3126
2	AdditiveProcessNoise(15)	0.3098
2	AdditiveProcessNoise(29)	0.3018
2	AdditiveProcessNoise(43)	0.2988
2	AdditiveProcessNoise(57)	0.2851
2	AdditiveProcessNoise(71)	0.2784
2	AdditiveProcessNoise(85)	0.2760
2	AdditiveProcessNoise(99)	0.2744
2	AdditiveProcessNoise(113)	0.2744
2	AdditiveProcessNoise(127)	0.2743
2	AdditiveProcessNoise(141)	0.2602
2	AdditiveProcessNoise(155)	0.2537
2	AdditiveProcessNoise(169)	0.2527
2	AccelerometerNoise	0.2524
2	GyroscopeNoise	0.2524
3	AdditiveProcessNoise(1)	0.2476
3	AdditiveProcessNoise(15)	0.2432
3	AdditiveProcessNoise(29)	0.2397
3	AdditiveProcessNoise(43)	0.2381
3	AdditiveProcessNoise(57)	0.2255
3	AdditiveProcessNoise(71)	0.2226
3	AdditiveProcessNoise(85)	0.2221
3	AdditiveProcessNoise(99)	0.2202
3	AdditiveProcessNoise(113)	0.2201
3	AdditiveProcessNoise(127)	0.2201
3	AdditiveProcessNoise(141)	0.2090
3	AdditiveProcessNoise(155)	0.2070
3	AdditiveProcessNoise(169)	0.2058
3	AccelerometerNoise	0.2052
3	GyroscopeNoise	0.2052
4	AdditiveProcessNoise(1)	0.2051
4	AdditiveProcessNoise(15)	0.2027
4	AdditiveProcessNoise(29)	0.2019
4	AdditiveProcessNoise(43)	0.2000
4	AdditiveProcessNoise(57)	0.1909
4	AdditiveProcessNoise(71)	0.1897
4	AdditiveProcessNoise(85)	0.1882
4	AdditiveProcessNoise(99)	0.1871
4	AdditiveProcessNoise(113)	0.1870
4	AdditiveProcessNoise(127)	0.1870
4	AdditiveProcessNoise(141)	0.1791
4	AdditiveProcessNoise(155)	0.1783
4	AdditiveProcessNoise(169)	0.1751
4	AccelerometerNoise	0.1748
4	GyroscopeNoise	0.1747
5	AdditiveProcessNoise(1)	0.1742
5	AdditiveProcessNoise(15)	0.1732
5	AdditiveProcessNoise(29)	0.1712

5	AdditiveProcessNoise(43)	0.1712
5	AdditiveProcessNoise(57)	0.1626
5	AdditiveProcessNoise(71)	0.1615
5	AdditiveProcessNoise(85)	0.1598
5	AdditiveProcessNoise(99)	0.1590
5	AdditiveProcessNoise(113)	0.1589
5	AdditiveProcessNoise(127)	0.1589
5	AdditiveProcessNoise(141)	0.1517
5	AdditiveProcessNoise(155)	0.1508
5	AdditiveProcessNoise(169)	0.1476
5	AccelerometerNoise	0.1473
5	GyroscopeNoise	0.1470
6	AdditiveProcessNoise(1)	0.1470
6	AdditiveProcessNoise(15)	0.1470
6	AdditiveProcessNoise(29)	0.1463
6	AdditiveProcessNoise(43)	0.1462
6	AdditiveProcessNoise(57)	0.1367
6	AdditiveProcessNoise(71)	0.1360
6	AdditiveProcessNoise(85)	0.1360
6	AdditiveProcessNoise(99)	0.1350
6	AdditiveProcessNoise(113)	0.1350
6	AdditiveProcessNoise(127)	0.1350
6	AdditiveProcessNoise(141)	0.1289
6	AdditiveProcessNoise(155)	0.1288
6	AdditiveProcessNoise(169)	0.1262
6	AccelerometerNoise	0.1253
6	GyroscopeNoise	0.1246
7	AdditiveProcessNoise(1)	0.1246
7	AdditiveProcessNoise(15)	0.1244
7	AdditiveProcessNoise(29)	0.1205
7	AdditiveProcessNoise(43)	0.1203
7	AdditiveProcessNoise(57)	0.1125
7	AdditiveProcessNoise(71)	0.1122
7	AdditiveProcessNoise(85)	0.1117
7	AdditiveProcessNoise(99)	0.1106
7	AdditiveProcessNoise(113)	0.1104
7	AdditiveProcessNoise(127)	0.1104
7	AdditiveProcessNoise(141)	0.1058
7	AdditiveProcessNoise(155)	0.1052
7	AdditiveProcessNoise(169)	0.1035
7	AccelerometerNoise	0.1024
7	GyroscopeNoise	0.1014
8	AdditiveProcessNoise(1)	0.1014
8	AdditiveProcessNoise(15)	0.1012
8	AdditiveProcessNoise(29)	0.1012
8	AdditiveProcessNoise(43)	0.1005
8	AdditiveProcessNoise(57)	0.0948
8	AdditiveProcessNoise(71)	0.0948
8	AdditiveProcessNoise(85)	0.0938
8	AdditiveProcessNoise(99)	0.0934
8	AdditiveProcessNoise(113)	0.0931
8	AdditiveProcessNoise(127)	0.0931
8	AdditiveProcessNoise(141)	0.0896
8	AdditiveProcessNoise(155)	0.0889
8	AdditiveProcessNoise(169)	0.0867
8	AccelerometerNoise	0.0859
8	GyroscopeNoise	0.0851
9	AdditiveProcessNoise(1)	0.0851

```

9      AdditiveProcessNoise(15)    0.0850
9      AdditiveProcessNoise(29)    0.0824
9      AdditiveProcessNoise(43)    0.0819
9      AdditiveProcessNoise(57)    0.0771
9      AdditiveProcessNoise(71)    0.0771
9      AdditiveProcessNoise(85)    0.0762
9      AdditiveProcessNoise(99)    0.0759
9      AdditiveProcessNoise(113)   0.0754
9      AdditiveProcessNoise(127)   0.0754
9      AdditiveProcessNoise(141)   0.0734
9      AdditiveProcessNoise(155)   0.0724
9      AdditiveProcessNoise(169)   0.0702
9      AccelerometerNoise          0.0697
9      GyroscopeNoise              0.0689
10     AdditiveProcessNoise(1)     0.0689
10     AdditiveProcessNoise(15)    0.0686
10     AdditiveProcessNoise(29)    0.0658
10     AdditiveProcessNoise(43)    0.0655
10     AdditiveProcessNoise(57)    0.0622
10     AdditiveProcessNoise(71)    0.0620
10     AdditiveProcessNoise(85)    0.0616
10     AdditiveProcessNoise(99)    0.0615
10     AdditiveProcessNoise(113)   0.0607
10     AdditiveProcessNoise(127)   0.0606
10     AdditiveProcessNoise(141)   0.0590
10     AdditiveProcessNoise(155)   0.0578
10     AdditiveProcessNoise(169)   0.0565
10     AccelerometerNoise          0.0562
10     GyroscopeNoise              0.0557
    
```

```
filter.AdditiveProcessNoise
```

```
ans = 13×13
```

```

0.5849    0    0    0    0    0    0    0    0
0    0.6484    0    0    0    0    0    0    0
0    0    0.5634    0    0    0    0    0    0
0    0    0    1.4271    0    0    0    0    0
0    0    0    0    4.3574    0    0    0    0
0    0    0    0    0    2.9527    0    0    0
0    0    0    0    0    0    1.3071    0    0
0    0    0    0    0    0    0    4.3574    0
0    0    0    0    0    0    0    0    2.2415
0    0    0    0    0    0    0    0    0
:
    
```

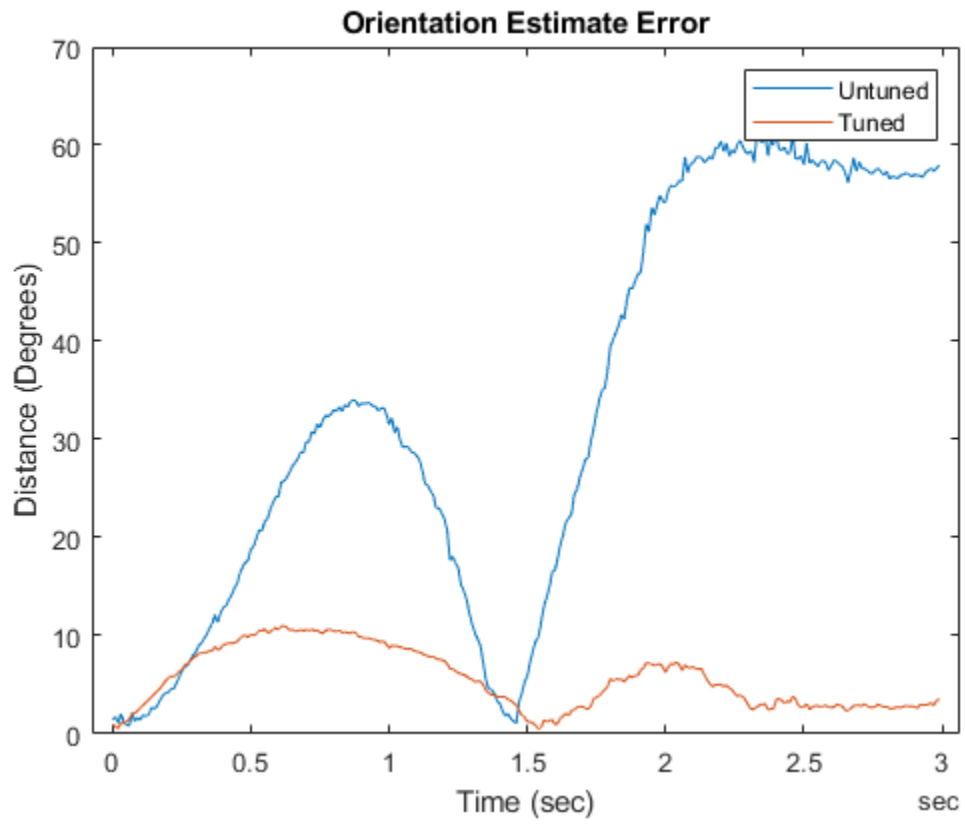
Batch-process the sensor data using the `estimateStates` object function. You can also recursively call the `predict` and `fuse` object functions to obtain the same results.

```
tunedEst = estimateStates(filter,ld.sensorData,tunedmn);
```

Compare the tuned and untuned estimates against the truth data. The results indicate that the tuning process greatly reduces the estimate errors.

```

distanceTuned = rad2deg(dist(tunedEst.Orientation,ld.groundTruth.Orientation));
hold on
plot(times,distanceTuned)
legend("Untuned","Tuned")
    
```



Occupancy Grids

In this section...

“Overview” on page 2-45

“World, Grid, and Local Coordinates” on page 2-45

“Inflation of Coordinates” on page 2-46

“Log-Odds Representation of Probability Values” on page 2-50

Overview

Occupancy grids are used to represent a robot workspace as a discrete grid. Information about the environment can be collected from sensors in real time or be loaded from prior knowledge. Laser range finders, bump sensors, cameras, and depth sensors are commonly used to find obstacles in your robot’s environment.

Occupancy grids are used in robotics algorithms such as path planning (see `mobileRobotPRM` or `plannerRRT`). They are used in mapping applications for integrating sensor information in a discrete map, in path planning for finding collision-free paths, and for localizing robots in a known environment (see `monteCarloLocalization` or `matchScans`). You can create maps with different sizes and resolutions to fit your specific application.

For 3-D occupancy maps, see `occupancyMap3D`.

For 2-D occupancy grids, there are two representations:

- Binary occupancy grid (see `binaryOccupancyMap`)
- Probability occupancy grid (see `occupancyMap`)

A binary occupancy grid uses `true` values to represent the occupied workspace (obstacles) and `false` values to represent the free workspace. This grid shows where obstacles are and whether a robot can move through that space. Use a binary occupancy grid if memory size is a factor in your application.

A probability occupancy grid uses probability values to create a more detailed map representation. This representation is the preferred method for using occupancy grids. This grid is commonly referred to as simply an occupancy grid. Each cell in the occupancy grid has a value representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free. The probabilistic values can give better fidelity of objects and improve performance of certain algorithm applications.

Binary and probability occupancy grids share several properties and algorithm details. Grid and world coordinates apply to both types of occupancy grids. The inflation function also applies to both grids, but each grid implements it differently. The effects of the log-odds representation and probability saturation apply to probability occupancy grids only.

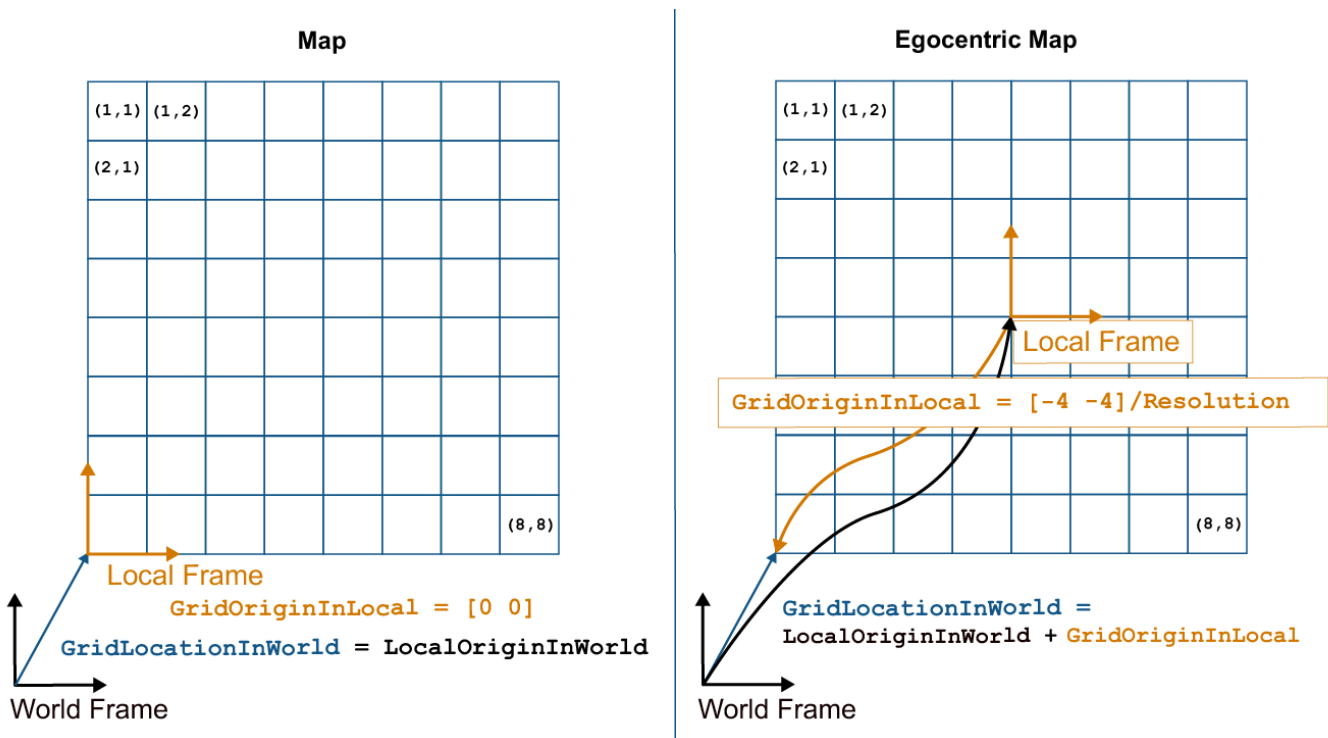
World, Grid, and Local Coordinates

When working with occupancy grids in MATLAB, you can use either world, local, or grid coordinates.

The absolute reference frame in which the robot operates is referred to as the world frame in the occupancy grid. Most operations are performed in the world frame, and it is the default selection when using MATLAB functions in this toolbox. World coordinates are used as an absolute coordinate frame with a fixed origin, and points can be specified with any resolution. However, all locations are converted to grid locations because of data storage and resolution limits on the map itself.

The local frame refers to the egocentric frame for a vehicle navigating the map. The `GridOriginInLocal` and `LocalOriginInWorld` properties define the origin of the grid in local coordinates and the relative location of the local frame in the world coordinates. You can adjust this local frame using the `move` function. For an example using the local frame as an egocentric map to emulate a vehicle moving around and sending local obstacles, see “Create Egocentric Occupancy Maps Using Range Sensors” on page 1-295.

Grid coordinates define the actual resolution of the occupancy grid and the finite locations of obstacles. The origin of grid coordinates is in the top-left corner of the grid, with the first location having an index of (1, 1). However, the `GridLocationInWorld` property of the occupancy grid in MATLAB defines the bottom-left corner of the grid in world coordinates. When creating an occupancy grid object, properties such as `XWorldLimits` and `YWorldLimits` are defined by the input width, height, and resolution. This figure shows a visual representation of these properties and the relation between world and grid coordinates.



Inflation of Coordinates

Both the binary and normal occupancy grids have an option for inflating obstacles. This inflation is used to add a factor of safety on obstacles and create buffer zones between the robot and obstacle in the environment. The `inflate` function of an occupancy grid object converts the specified radius to the number of cells rounded up from the `resolution*radius` value. Each algorithm uses this cell value separately to modify values around obstacles.

Binary Occupancy Grid

The `inflate` function takes each occupied cell and directly inflates it by adding occupied space around each point. This basic inflation example illustrates how the radius value is used.

Inflate Obstacles in a Binary Occupancy Grid

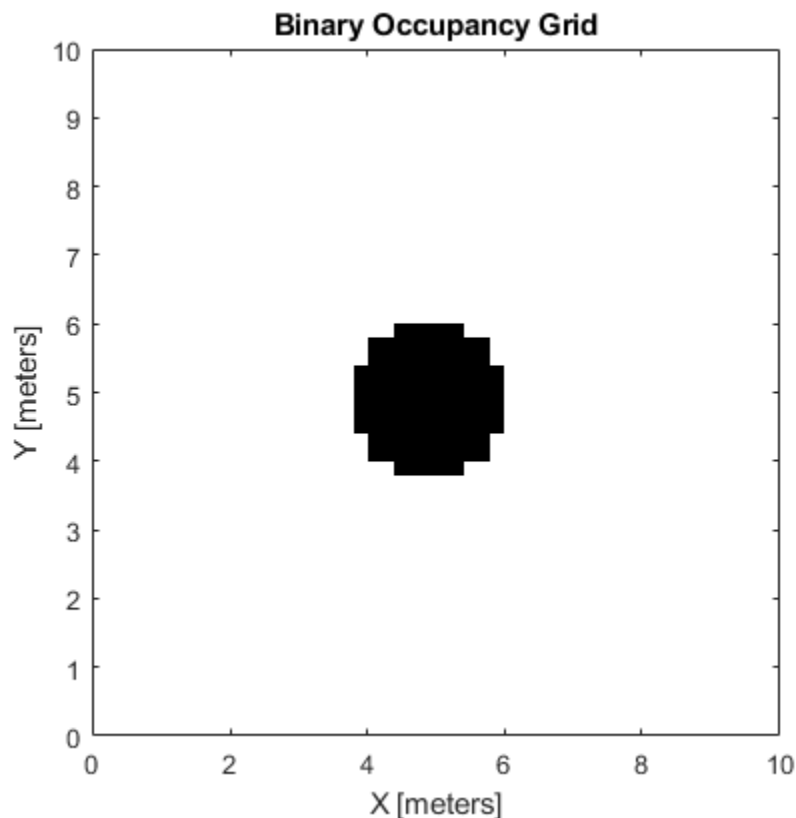
This example shows how to create the map, set the obstacle locations and inflate it by a radius of 1m. Extra plots on the figure help illustrate the inflation and shifting due to conversion to grid locations.

Create binary occupancy grid. Set occupancy of position [5,5].

```
map = binaryOccupancyMap(10,10,5);
setOccupancy(map,[5 5], 1);
```

Inflate occupied spaces on map by 1m.

```
inflate(map,1);
show(map)
```



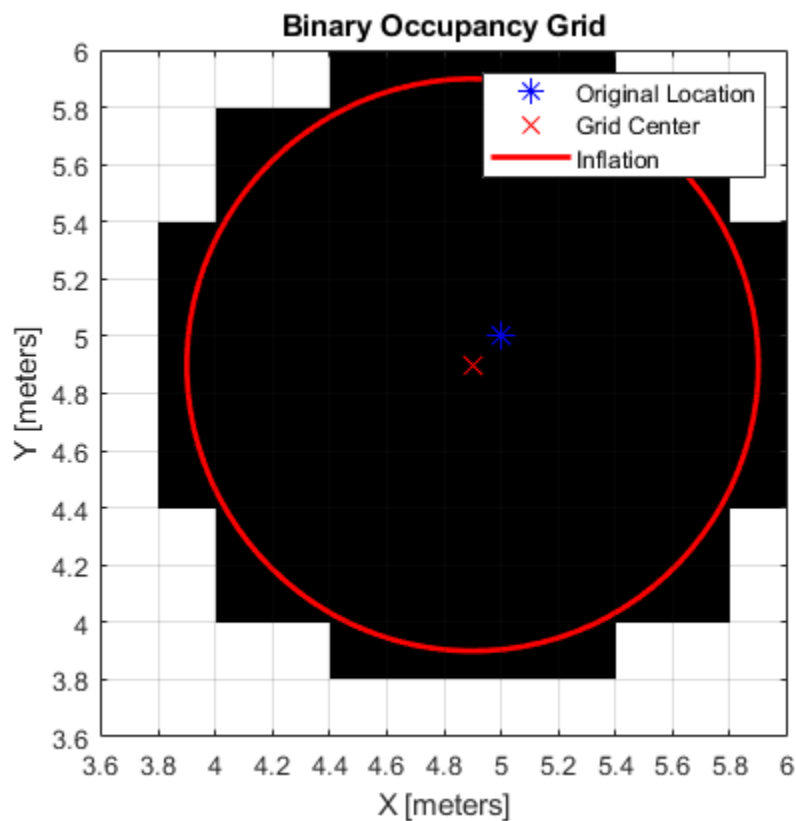
Plot original location, converted grid position and draw the original circle. You can see from this plot, that the grid center is [4.9 4.9], which is shifted from the [5 5] location. A 1m circle is drawn from there and notice that any cells that touch this circle are marked as occupied. The figure is zoomed in to the relevant area.

```
hold on
theta = linspace(0,2*pi);
```

```

x = 4.9+cos(theta); % x circle coordinates
y = 4.9+sin(theta); % y circle coordinates
plot(5,5,'*b','MarkerSize',10) % Original location
plot(4.9,4.9,'xr','MarkerSize',10) % Grid location center
plot(x,y,'-r','LineWidth',2); % Circle of radius 1m.
axis([3.6 6 3.6 6])
ax = gca;
ax.XTick = [3.6:0.2:6];
ax.YTick = [3.6:0.2:6];
grid on
legend('Original Location','Grid Center','Inflation')

```



As you can see from the above figure, even cells that barely overlap with the inflation radius are labeled as occupied.

Occupancy Grids

The `inflate` function uses the inflation radius to perform *probabilistic inflation*. Probabilistic inflation acts as a local maximum operator and finds the highest probability values for nearby cells. The `inflate` function uses this definition to inflate the higher probability values throughout the grid. This inflation increases the size of any occupied locations and creates a buffer zone for robots to navigate around obstacles. This example shows how the inflation works with a range of probability values.

Inflate Obstacles in an Occupancy Grid

This example shows how the `inflate` method performs probabilistic inflation on obstacles to inflate their size and create a buffer zone for areas with a higher probability of obstacles.

Create a 10m x 10m empty map.

```
map = occupancyMap(10,10,10);
```

Update occupancy of world locations with specific values in `pvalues`.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
```

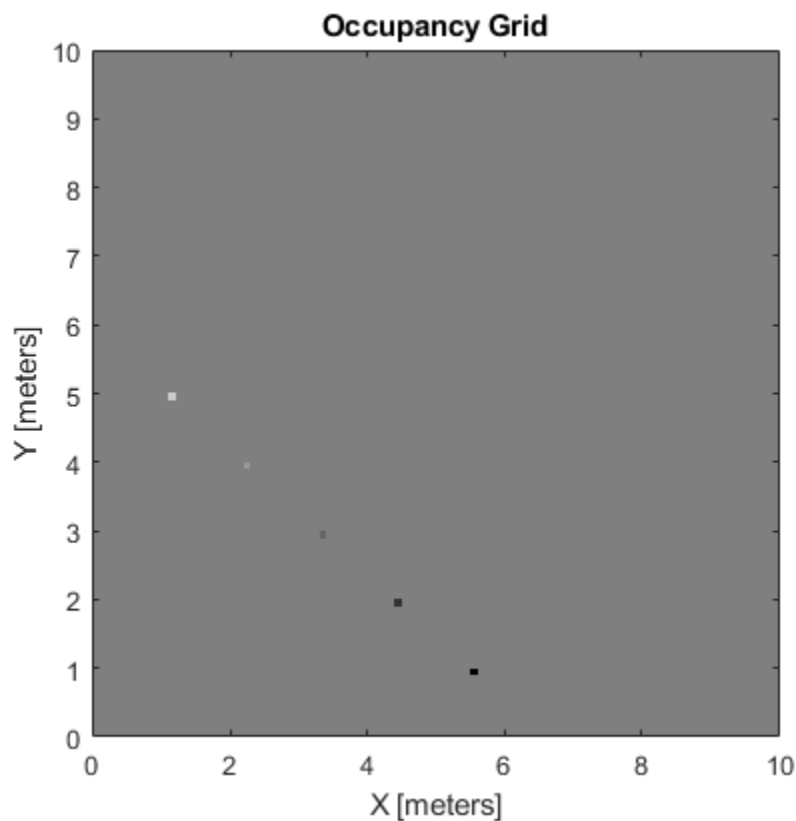
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
pvalues = [0.2 0.4 0.6 0.8 1];
```

```
updateOccupancy(map,[x y],pvalues)
```

```
figure
```

```
show(map)
```



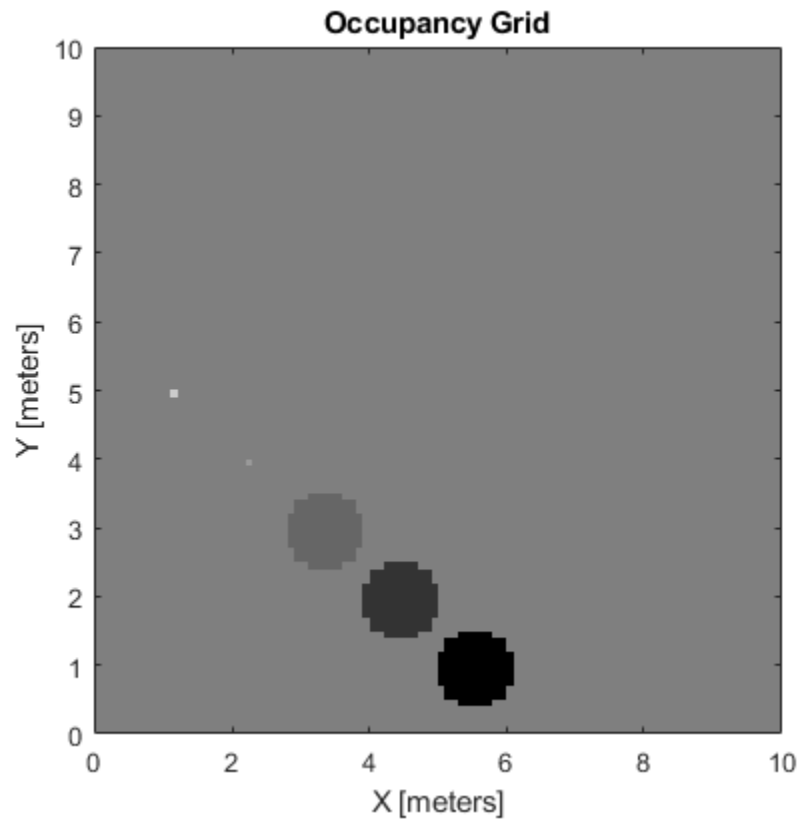
Inflate occupied areas by a given radius. Larger occupancy values are written over smaller values. You can copy your map beforehand to revert any unwanted changes.

```
savedMap = copy(map);
```

```
inflate(map,0.5)
```

```
figure
```

```
show(map)
```

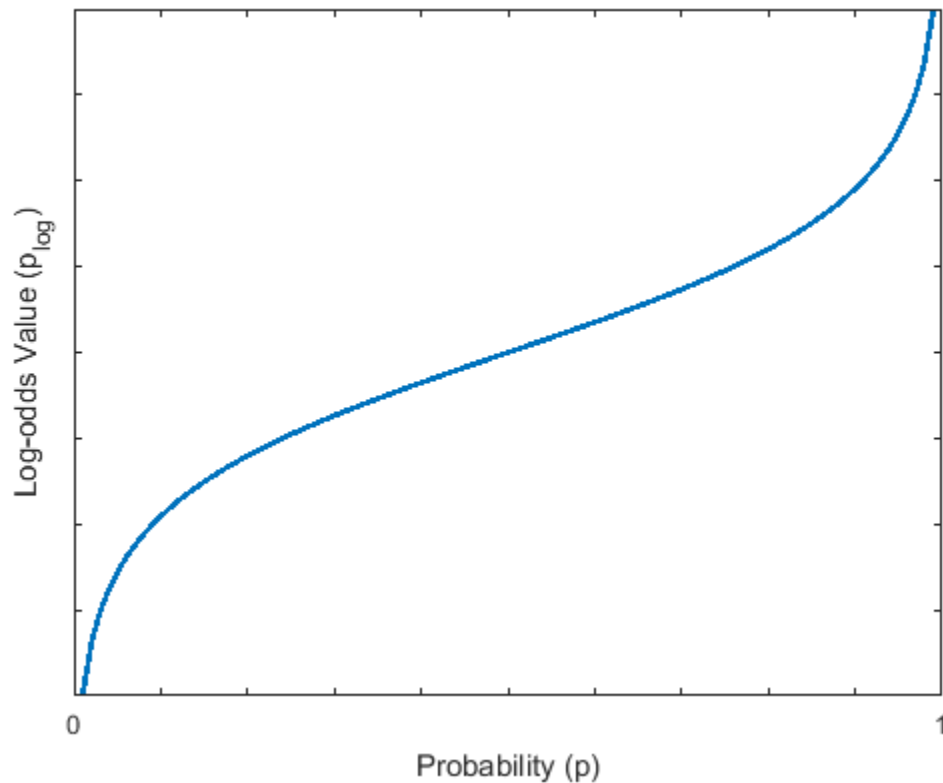


Log-Odds Representation of Probability Values

When using occupancy grids with probability values, the goal is to estimate the probability of obstacle locations for use in real-time robotics applications. The `occupancyMap` class uses a *log-odds* representation of the probability values for each cell. Each probability value is converted to a corresponding log-odds value for internal storage. The value is converted back to probability when accessed. This representation efficiently updates probability values with the fewest operations. Therefore, you can quickly integrate sensor data into the map.

The log-odds representation uses the following equation:

$$p_{\log} = \log\left(\frac{p}{1-p}\right)$$



Note Log-odds values are stored as `int16` values. This data type limits the resolution of probability values to ± 0.001 but greatly improves memory size and allows for creation of larger maps.

Probability Saturation

When updating an occupancy grid with observations using the log-odds representation, the values have a range of $-\infty$ to ∞ . This range means if a robot observes a location such as a closed door multiple times, the log-odds value for this location becomes unnecessarily high, or the value probability gets saturated. If the door then opens, the robot needs to observe the door open many times before the probability changes from occupied to free. In dynamic environments, you want the map to react to changes to more accurately track dynamic objects.

To prevent this saturation, update the `ProbabilitySaturation` property, which limits the minimum and maximum probability values allowed when incorporating multiple observations. This property is an upper and lower bound on the log-odds values and enables the map to update quickly to changes in the environment. The default minimum and maximum values of the saturation limits are `[0.001 0.999]`. For dynamic environments, the suggested values are at least `[0.12 0.97]`. Consider modifying this range if the map does not update rapidly enough for multiple observations.

See Also

`binaryOccupancyMap` | `occupancyMap` | `occupancyMap3D`

Related Examples

- “Create Egocentric Occupancy Maps Using Range Sensors” on page 1-295
- “Build Occupancy Map from Lidar Scans and Poses” on page 1-300

Choose Path Planning Algorithms for Navigation

The Navigation Toolbox provides multiple path or motion planners to generate a sequence of valid configurations that move an object from a start to an end goal. The toolbox supports both global and local planners. Global planners typically require a map and define the overall state space. Local planners typically take a globally planned path and adjust the path based on obstacles in the environment. Planners check for collisions with the environment, connect and propagate states, and use cost functions for optimality. The table below details the key differences between the different planners and when to use a certain one.

Planner	Type and Scope	Collision Checking	State Connection and Propagation	Benefits	Used For
Grid-Based A* — planner AStarGrid	Global path planner	Occupancy map (validatorOccupancyMap)	Connection: XY linear motion primitives Propagation: Not supported	<ul style="list-style-type: none"> • Customizable cost and heuristics • Optimality if heuristics are consistent and admissible 	Omnidrive robots
Hybrid A* — planner HybridAStar	Global path planner	Occupancy map (validatorOccupancyMap or validatorVehicleCostmap)	Connection: Reeds-Shepp motion primitive Propagation: Circular arc motion primitive	<ul style="list-style-type: none"> • Differential constraints for state propagation 	Nonholonomic vehicles with a minimum turning radius
Rapidly-exploring Random Tree (RRT) — planner RRT	Global path planner	General state validator	Connection: General state space Propagation: Not supported	<ul style="list-style-type: none"> • Customizable 	Manipulators, omnidrive robots, vehicles with a minimum turning radius
RRT* — planner RRTStar	Global path planner	General state validator	Connection: General state space Propagation: Not supported	<ul style="list-style-type: none"> • Customizable • Asymptotically optimal 	Manipulators, omnidrive robots, vehicles with a minimum turning radius
Frenet Trajectory — trajectoryOptimalFrenet	Local trajectory generator	General state validator	Connection: Quintic polynomials or clothoids Propagation: Not applicable	<ul style="list-style-type: none"> • Customizable collision checking • Self-defined optimality 	Ackerman type vehicles for highway driving

See Also

Related Examples

- “Plan Mobile Robot Paths Using RRT” on page 1-341
- “Dynamic Replanning on an Indoor Map” on page 1-372
- “Highway Lane Change” on page 1-378
- “Highway Trajectory Planning Using Frenet Reference Path” on page 1-397

Execute Code at a Fixed-Rate

In this section...

“Introduction” on page 2-55

“Run Loop at Fixed Rate” on page 2-55

“Overrun Actions for Fixed Rate Execution” on page 2-55

Introduction

By executing code at constant intervals, you can accurately time and schedule tasks. Using a `rateControl` object allows you to control the rate of your code execution. These examples show different applications for the `rateControl` object including its uses with ROS and sending commands for robot control.

Run Loop at Fixed Rate

Create a rate object that runs at 1 Hz.

```
r = rateControl(1);
```

Start a loop using the `rateControl` object inside to control the loop execution. Reset the object prior to the loop execution to reset timer. Print the iteration and time elapsed.

```
reset(r)
for i = 1:10
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    waitfor(r);
end
```

```
Iteration: 1 - Time Elapsed: 0.002257
Iteration: 2 - Time Elapsed: 1.011465
Iteration: 3 - Time Elapsed: 2.011186
Iteration: 4 - Time Elapsed: 3.014202
Iteration: 5 - Time Elapsed: 4.001980
Iteration: 6 - Time Elapsed: 5.001219
Iteration: 7 - Time Elapsed: 6.000373
Iteration: 8 - Time Elapsed: 7.000181
Iteration: 9 - Time Elapsed: 8.000670
Iteration: 10 - Time Elapsed: 9.000240
```

Each iteration executes at a 1-second interval.

Overrun Actions for Fixed Rate Execution

The `rateControl` object uses the `OverrunAction` property to decide how to handle code that takes longer than the desired period to operate. The options are 'slip' (default) or 'drop'. This example shows how the `OverrunAction` affects code execution.

Setup desired rate and loop time. `slowFrames` is an array of times when the loop should be stalled longer than the desired rate.

```
desiredRate = 1;  
loopTime = 20;  
slowFrames = [3 7 12 18];
```

Create the Rate object and specify the `OverrunAction` property. 'slip' indicates that the `waitfor` function will return immediately if the time for `LastPeriod` is greater than the `DesiredRate` property.

```
rate = rateControl(desiredRate);  
rate.OverrunAction = 'slip';
```

Reset Rate object and begin loop. This loop will execute at the desired rate until the loop time is reached. When the `TotalElapsedTime` reaches a slow frame time, it will stall for longer than the desired period.

```
reset(rate);  
  
while rate.TotalElapsedTime < loopTime  
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))  
        pause(desiredRate + 0.1)  
    end  
    waitfor(rate);  
end
```

View statistics on the Rate object. Notice the number of periods.

```
stats = statistics(rate)  
  
stats = struct with fields:  
    Periods: [1.0143 0.9965 1.0014 1.1026 1.0077 0.9991 0.9961 ... ]  
    NumPeriods: 20  
    AveragePeriod: 1.0225  
    StandardDeviation: 0.0421  
    NumOverruns: 4
```

Change the `OverrunAction` to 'drop'. 'drop' indicates that the `waitfor` function will return at the next time step, even if the `LastPeriod` is greater than the `DesiredRate` property. This effectively drops the iteration that was missed by the slower code execution.

```
rate.OverrunAction = 'drop';
```

Reset Rate object and begin loop.

```
reset(rate);  
  
while rate.TotalElapsedTime < loopTime  
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))  
        pause(1.1)  
    end  
    waitfor(rate);  
end  
stats2 = statistics(rate)  
  
stats2 = struct with fields:  
    Periods: [1.0143 0.9872 1.0096 1.9941 1.0006 0.9991 2.0073 ... ]  
    NumPeriods: 16  
    AveragePeriod: 1.2501
```

```
StandardDeviation: 0.4481  
NumOVERRUNS: 4
```

Using the 'drop' over run action resulted in 16 periods when the 'slip' resulted in 20 periods. This difference is because the 'slip' did not wait until the next interval based on the desired rate. Essentially, using 'slip' tries to keep the `AveragePeriod` property as close to the desired rate. Using 'drop' ensures the code will execute at an even interval relative to `DesiredRate` with some iterations being skipped.

See Also

`rateControl` | `rosclock` | `waitfor`

Particle Filter Workflow

A particle filter is a recursive, Bayesian state estimator that uses discrete particles to approximate the posterior distribution of the estimated state.

The particle filter algorithm computes the state estimate recursively and involves two steps:

- Prediction - The algorithm uses the previous state to predict the current state based on a given system model.
- Correction - The algorithm uses the current sensor measurement to correct the state estimate.

The algorithm also periodically redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state.

The estimated state consists of all the state variables. Each particle represents a discrete state hypothesis. The set of all particles is used to help determine the final state estimate.

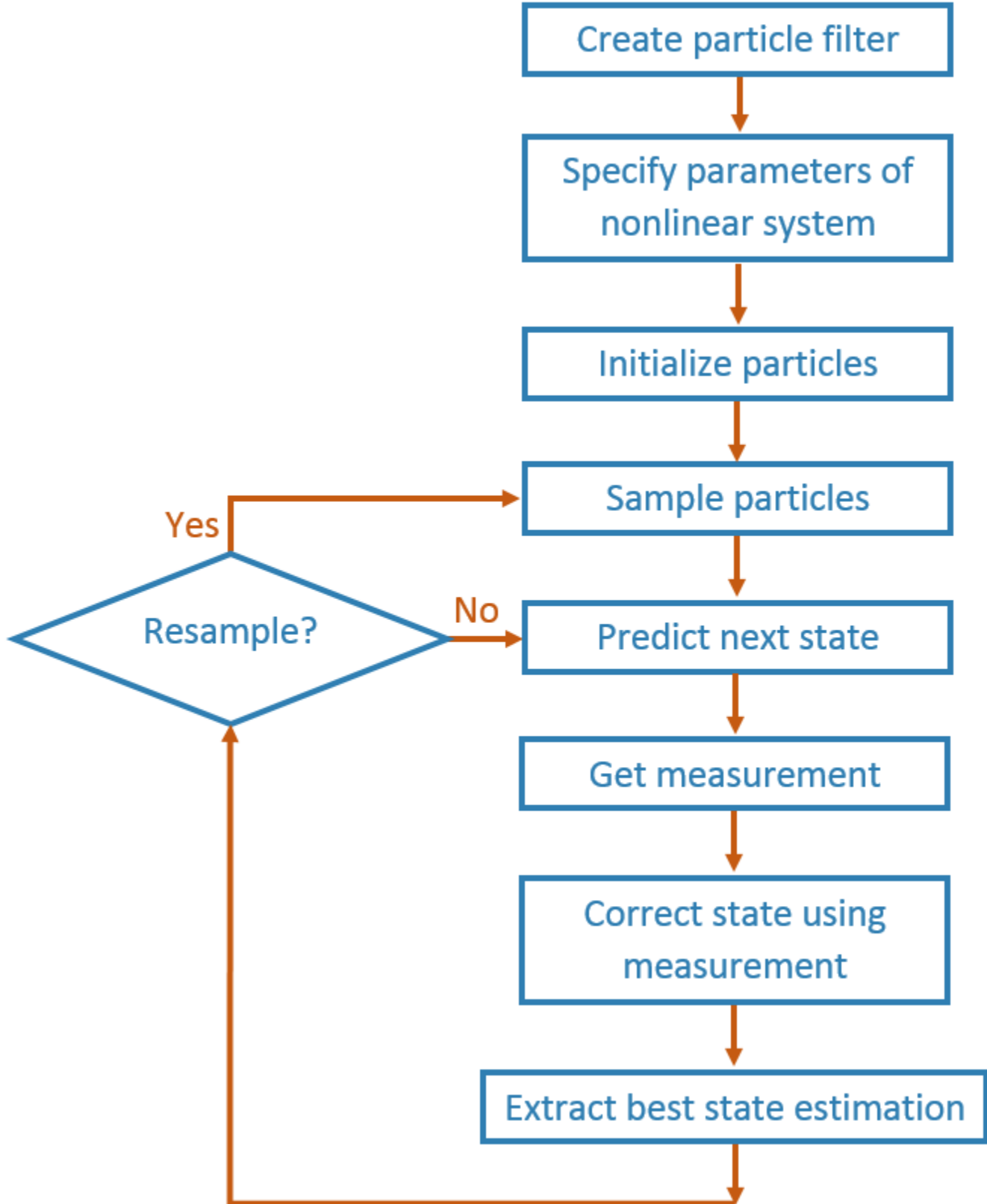
You can apply the particle filter to arbitrary nonlinear system models. Process and measurement noise can follow arbitrary non-Gaussian distributions.

To use the particle filter properly, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement likelihood function, respectively. For more information, see “Particle Filter Parameters” on page 2-62.

Follow this basic workflow to create and use a particle filter. This page details the estimation workflow and shows an example of how to run a particle filter in a loop to continuously estimate state.

Estimation Workflow

When using a particle filter, there is a required set of steps to create the particle filter and estimate state. The prediction and correction steps are the main iteration steps for continuously estimating state.



Create Particle Filter

Create a `stateEstimatorPF` object.

Set Parameters of Nonlinear System

Modify these `stateEstimatorPF` parameters to fit for your specific system or application:

- `StateTransitionFcn`
- `MeasurementLikelihoodFcn`
- `ResamplingPolicy`
- `ResamplingMethod`
- `StateEstimationMethod`

Default values for these parameters are given for basic operation.

The `StateTransitionFcn` and `MeasurementLikelihoodFcn` functions define the system behavior and measurement integration. They are vital for the particle filter to track accurately. For more information, see “Particle Filter Parameters” on page 2-62.

Initialize Particles

Use the `initialize` function to set the number of particles and the initial state.

Sample Particles from a Distribution

You can sample the initial particle locations in two ways:

- Initial pose and covariance — If you have an idea of your initial state, it is recommended you specify the initial pose and covariance. This specification helps to cluster particles closer to your estimate so tracking is more effective from the start.
- State bounds — If you do not know your initial state, you can specify the possible limits of each state variable. Particles are uniformly distributed across the state bounds for each variable. Widely distributed particles are not as effective at tracking, because fewer particles are near the actual state. Using state bounds usually requires more particles, computation time, and iterations to converge to the actual state estimate.

Predict

Based on a specified state transition function, particles evolve to estimate the next state. Use `predict` to execute the state transition function specified in the `StateTransitionFcn` property.

Get Measurement

The measurements collected from sensors are used in the next step to correct the current predicted state.

Correct

Measurements are then used to adjust the predicted state and correct the estimate. Specify your measurements using the `correct` function. `correct` uses the `MeasurementLikelihoodFcn` to calculate the likelihood of sensor measurements for each particle. Resampling of particles is required to update your estimation as the state changes in subsequent iterations. This step triggers resampling based on the `ResamplingMethod` and `ResamplingPolicy` properties.

Extract Best State Estimation

After calling `correct`, the best state estimate is automatically extracted based on the `Weights` of each particle and the `StateEstimationMethod` property specified in the object. The best estimated state and covariance is output by the `correct` function.

Resample Particles

This step is not separately called, but is executed when you call `correct`. Once your state has changed enough, resample your particles based on the newest estimate. The `correct` method checks the `ResamplingPolicy` for the triggering of particle resampling according to the current distribution of particles and their weights. If resampling is not triggered, the same particles are used for the next estimation. If your state does not vary by much or if your time step is low, you can call the `predict` and `correct` methods without resampling.

Continuously Predict and Correct

Repeat the previous prediction and correction steps as needed for estimating state. The correction step determines if resampling of the particles is required. Multiple calls for `predict` or `correct` might be required when:

- No measurement is available but control inputs and time updates are occur at a high frequency. Use the `predict` method to evolve the particles to get the updated predicted state more often.
- Multiple measurement reading are available. Use `correct` to integrate multiple readings from the same or multiple sensors. The function corrects the state based on each set of information collected.

See Also

`stateEstimatorPF` | `initialize` | `getStateEstimate` | `predict` | `correct`

Related Examples

- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Parameters” on page 2-62

Particle Filter Parameters

In this section...
“Number of Particles” on page 2-62
“Initial Particle Location” on page 2-63
“State Transition Function” on page 2-64
“Measurement Likelihood Function” on page 2-65
“Resampling Policy” on page 2-65
“State Estimation Method” on page 2-66

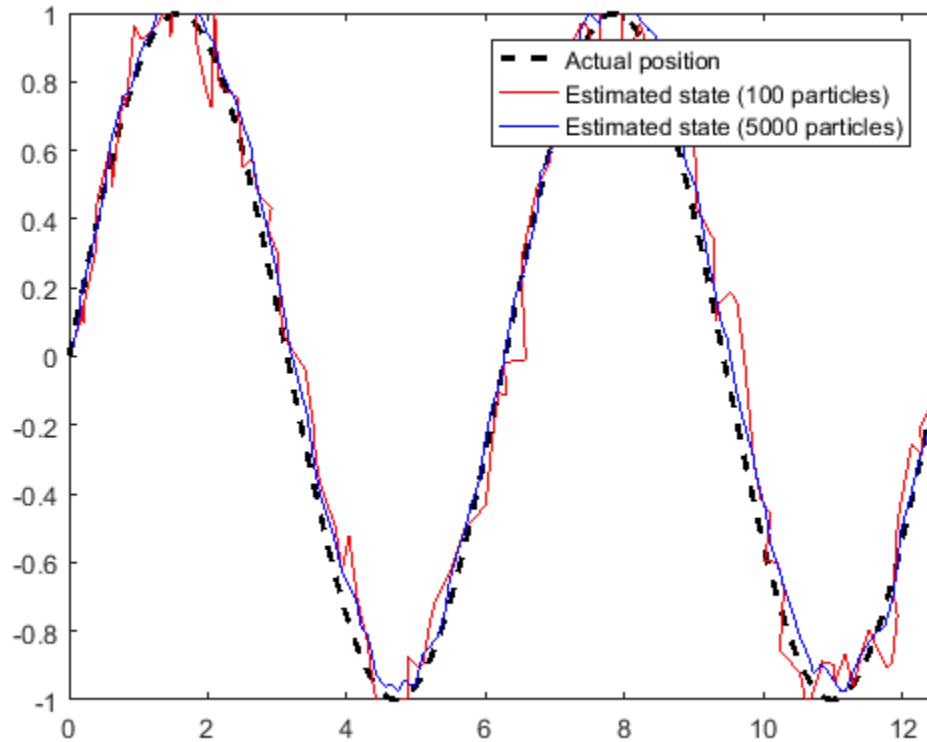
To use the `stateEstimatorPF` particle filter, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement likelihood function, respectively. The details of these parameters are detailed on this page. For more information on the particle filter workflow, see “Particle Filter Workflow” on page 2-58.

Number of Particles

To specify the number of particles, use the `initialize` method. Each particle is a hypothesis of the current state. The particles are distributed across your state space based on either a specified mean and covariance, or on the specified state bounds. Depending on the `StateEstimationMethod` property, either the particle with the highest weight or the mean of all particles is taken to determine the best state estimate.

The default number of particles is 1000. Unless performance is an issue, do not use fewer than 1000 particles. A higher number of particles can improve the estimate but sacrifices performance speed, because the algorithm has to process more particles. Tuning the number of particles is the best way to affect your particle filters performance.

These results, which are based on the `stateEstimatorPF` example, show the difference in tracking accuracy when using 100 particles and 5000 particles.



Initial Particle Location

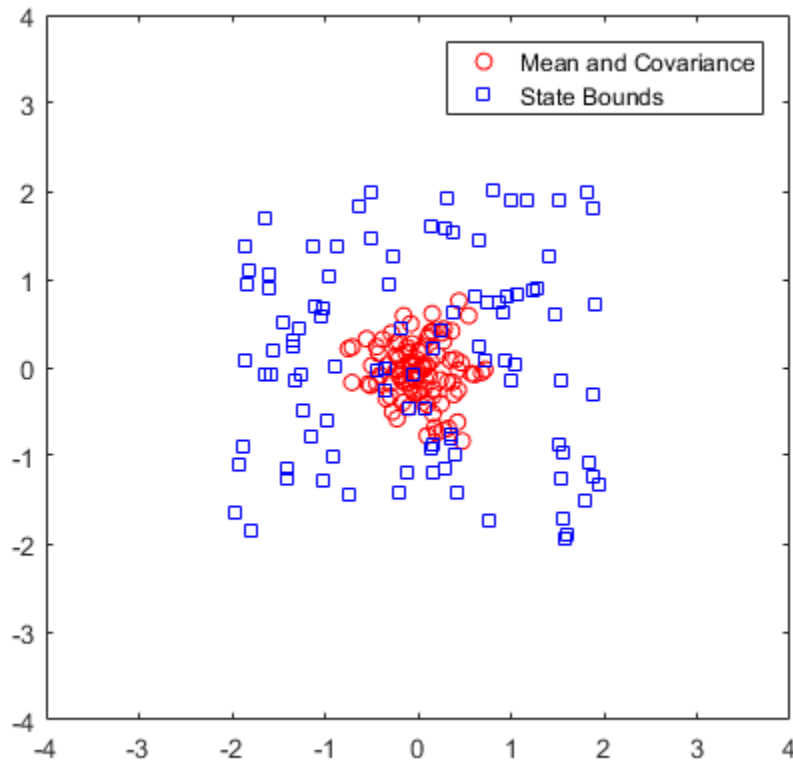
When you initialize your particle filter, you can specify the initial location of the particles using:

- Mean and covariance
- State bounds

Your initial state is defined as a mean with a covariance relative to your system. This mean and covariance correlate to the initial location and uncertainty of your system. The `stateEstimatorPF` object distributes particles based on your covariance around the given mean. The algorithm uses this distribution of particles to get the best estimation of state, so an accurate initialization of particles helps to converge to the best state estimation quickly.

If an initial state is unknown, you can evenly distribute your particles across a given state bounds. The state bounds are the limits of your state. For example, when estimating the position of a robot, the state bounds are limited to the environment that the robot can actually inhabit. In general, an even distribution of particles is a less efficient way to initialize particles to improve the speed of convergence.

The plot shows how the mean and covariance specification can cluster particles much more effectively in a space rather than specifying the full state bounds.



State Transition Function

The state transition function, `StateTransitionFcn`, of a particle filter helps to evolve the particles to the next state. It is used during the prediction step of the “Particle Filter Workflow” on page 2-58. In the `stateEstimatorPF` object, the state transition function is specified as a callback function that takes the previous particles, and any other necessary parameters, and outputs the predicted location. The function header syntax is:

```
function predictParticles = stateTransitionFcn(pf,prevParticles,varargin)
```

By default, the state transition function assumes a Gaussian motion model with constant velocities. The function uses a Gaussian distribution to determine the position of the particles in the next time step.

For your application, it is important to have a state transition function that accurately describes how you expect the system to behave. To accurately evolve all the particles, you must develop and implement a motion model for your system. If particles are not distributed around the next state, the `stateEstimatorPF` object does not find an accurate estimate. Therefore, it is important to understand how your system can behave so that you can track it accurately.

You also must specify system noise in `StateTransitionFcn`. Without random noise applied to the predicted system, the particle filter does not function as intended.

Although you can predict many systems based on their previous state, sometimes the system can include extra information. The use of `varargin` in the function enables you to input any extra

parameters that are relevant for predicting the next state. When you call `predict`, you can include these parameters using:

```
predict(pf,param1,param2)
```

Because these parameters match the state transition function you defined, calling `predict` essentially calls the function as:

```
predictParticles = stateTransitionFcn(pf,prevParticles,param1,param2)
```

The output particles, `predictParticles`, are then either used by the “Measurement Likelihood Function” on page 2-65 to correct the particles, or used in the next prediction step if correction is not required.

Measurement Likelihood Function

After predicting the next state, you can use measurements from sensors to correct your predicted state. By specifying a `MeasurementLikelihoodFcn` in the `stateEstimatorPF` object, you can correct your predicted particles using the `correct` function. This measurement likelihood function, by definition, gives a weight for the state hypotheses (your particles) based on a given measurement. Essentially, it gives you the likelihood that the observed measurement actually matches what each particle observes. This likelihood is used as a weight on the predicted particles to help with correcting them and getting the best estimation. Although the prediction step can prove accurate for a small number of intermediate steps, to get accurate tracking, use sensor observations to correct the particles frequently.

The specification of the `MeasurementLikelihoodFcn` is similar to the `StateTransitionFcn`. It is specified as a function handle in the properties of the `stateEstimatorPF` object. The function header syntax is:

```
function likelihood = measurementLikelihoodFcn(pf,predictParticles,measurement,varargin)
```

The output is the likelihood of each predicted particle based on the measurement given. However, you can also specify more parameters in `varargin`. The use of `varargin` in the function enables you to input any extra parameters that are relevant for correcting the predicted state. When you call `correct`, you can include these parameters using:

```
correct(pf,measurement,param1,param2)
```

These parameters match the measurement likelihood function you defined:

```
likelihood = measurementLikelihoodFcn(pf,predictParticles,measurement,param1,param2)
```

The `correct` function uses the `likelihood` output for particle resampling and giving the final state estimate.

Resampling Policy

The resampling of particles is a vital step for continuous tracking of objects. It enables you to select particles based on the current state, instead of using the particle distribution given at initialization. By continuously resampling the particles around the current estimate, you can get more accurate tracking and improve long-term performance.

When you call `correct`, the particles used for state estimation can be resampled depending on the `ResamplingPolicy` property specified in the `stateEstimatorPF` object. This property is specified

as a `resamplingPolicyPF` object. The `TriggerMethod` property on that object tells the particle filter which method to use for resampling.

You can trigger resampling at either a fixed interval or when a minimum effective particle ratio is reached. The fixed interval method resamples at a set number of iterations, which is specified in the `SamplingInterval` property. The minimum effective particle ratio is a measure of how well the current set of particles approximates the posterior distribution. The number of effective particles is calculated by:

$$N_{eff} = \frac{1}{\sum_{i=1}^N (w^i)^2}$$

In this equation, N is the number of particles, and w is the normalized weight of each particle. The effective particle ratio is then $N_{eff} / \text{NumParticles}$. Therefore, the effective particle ratio is a function of the weights of all the particles. After the weights of the particles reach a low enough value, they are not contributing to the state estimation. This low value triggers resampling, so the particles are closer to the current state estimation and have higher weights.

State Estimation Method

The final step of the particle filter workflow is the selection of a single state estimate. The particles and their weights sampled across the distribution are used to give the best estimation of the actual state. However, you can use the particles information to get a single state estimate in multiple ways. With the `stateEstimatorPF` object, you can either choose the best estimate based on the particle with the highest weight or take a mean of all the particles. Specify the estimation method in the `StateEstimationMethod` property as either 'mean'(default) or 'maxweight'.

Because you can estimate the state from all of the particles in many ways, you can also extract each particle and its weight from the `stateEstimatorPF` using the `Particles` property.

See Also

`stateEstimatorPF` | `resamplingPolicyPF`

Related Examples

- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Workflow” on page 2-58

Pure Pursuit Controller

In this section...

“Reference Coordinate System” on page 2-67

“Look Ahead Distance” on page 2-67

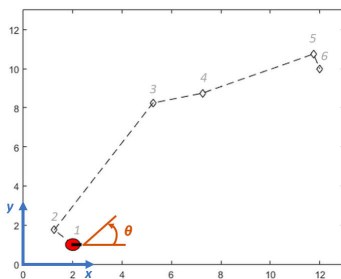
“Limitations” on page 2-68

Pure pursuit is a path tracking algorithm. It computes the angular velocity command that moves the robot from its current position to reach some look-ahead point in front of the robot. The linear velocity is assumed constant, hence you can change the linear velocity of the robot at any point. The algorithm then moves the look-ahead point on the path based on the current position of the robot until the last point of the path. You can think of this as the robot constantly chasing a point in front of it. The property `LookAheadDistance` decides how far the look-ahead point is placed.

The controller `PurePursuit` object is not a traditional controller, but acts as a tracking algorithm for path following purposes. Your controller is unique to a specified a list of waypoints. The desired linear and maximum angular velocities can be specified. These properties are determined based on the vehicle specifications. Given the pose (position and orientation) of the vehicle as an input, the object can be used to calculate the linear and angular velocities commands for the robot. How the robot uses these commands is dependent on the system you are using, so consider how robots can execute a motion given these commands. The final important property is the `LookAheadDistance`, which tells the robot how far along on the path to track towards. This property is explained in more detail in a section below.

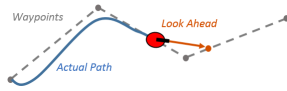
Reference Coordinate System

It is important to understand the reference coordinate frame used by the pure pursuit algorithm for its inputs and outputs. The figure below shows the reference coordinate system. The input waypoints are $[x \ y]$ coordinates, which are used to compute the robot velocity commands. The robot's pose is input as a pose and orientation (θ) list of points as $[x \ y \ \theta]$. The positive x and y directions are in the right and up directions respectively (blue in figure). The θ value is the angular orientation of the robot measured counterclockwise in radians from the x -axis (robot currently at θ radians).

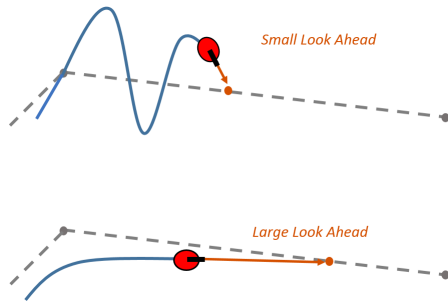


Look Ahead Distance

The `LookAheadDistance` property is the main tuning property for the controller. The look ahead distance is how far along the path the robot should look from the current location to compute the angular velocity commands. The figure below shows the robot and the look-ahead point. As displayed in this image, note that the actual path does not match the direct line between waypoints.



The effect of changing this parameter can change how your robot tracks the path and there are two major goals: regaining the path and maintaining the path. In order to quickly regain the path between waypoints, a small `LookAheadDistance` will cause your robot to move quickly towards the path. However, as can be seen in the figure below, the robot overshoots the path and oscillates along the desired path. In order to reduce the oscillations along the path, a larger look ahead distance can be chosen, however, it might result in larger curvatures near the corners.



The `LookAheadDistance` property should be tuned for your application and robot system. Different linear and angular velocities will affect this response as well and should be considered for the path following controller.

Limitations

There are a few limitations to note about this pure pursuit algorithm:

- As shown above, the controller cannot exactly follow direct paths between waypoints. Parameters must be tuned to optimize the performance and to converge to the path over time.
- This pure pursuit algorithm does not stabilize the robot at a point. In your application, a distance threshold for a goal location should be applied to stop the robot near the desired goal.

References

- [1] Coulter, R. *Implementation of the Pure Pursuit Path Tracking Algorithm*. Carnegie Mellon University, Pittsburgh, Pennsylvania, Jan 1990.

See Also

`stateEstimatorPF` | `controllerVFH`

Monte Carlo Localization Algorithm

In this section...

- "Overview" on page 2-69
- "State Representation" on page 2-69
- "Initialization of Particles" on page 2-71
- "Resampling Particles and Updating Pose" on page 2-73
- "Motion and Sensor Model" on page 2-74

Overview

The Monte Carlo Localization (MCL) algorithm is used to estimate the position and orientation of a robot. The algorithm uses a known map of the environment, range sensor data, and odometry sensor data. To see how to construct an object and use this algorithm, see `monteCarloLocalization`.

To localize the robot, the MCL algorithm uses a particle filter to estimate its position. The particles represent the distribution of the likely states for the robot. Each particle represents a possible robot state. The particles converge around a single location as the robot moves in the environment and senses different parts of the environment using a range sensor. The robot motion is sensed using an odometry sensor.

The particles are updated in this process:

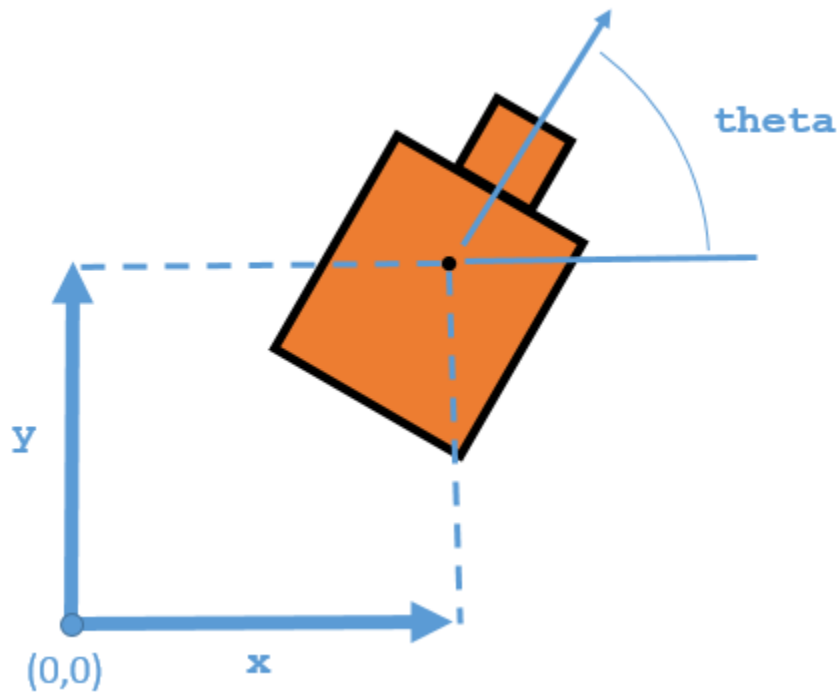
- 1 Particles are propagated based on the change in the pose and the specified motion model, `MotionModel`.
- 2 The particles are assigned weights based on the likelihood of receiving the range sensor reading for each particle. This reading is based on the sensor model you specify in `SensorModel`.
- 3 Based on these weights, a robot state estimate is extracted based on the particle weights. The group of particles with the highest weight is used to estimate the position of the robot.
- 4 Finally, the particles are resampled based on the specified `ResamplingInterval`. Resampling adjusts particle positions and improves performance by adjusting the number of particles used. It is a key feature for adjusting to changes and keeping particles relevant for estimating the robot state.

The algorithm outputs the estimated pose and covariance. These estimates are the mean and covariance of the highest weighted cluster of particles. For continuous tracking, repeat these steps in a loop to propagate particles, evaluate their likelihood, and get the best state estimate.

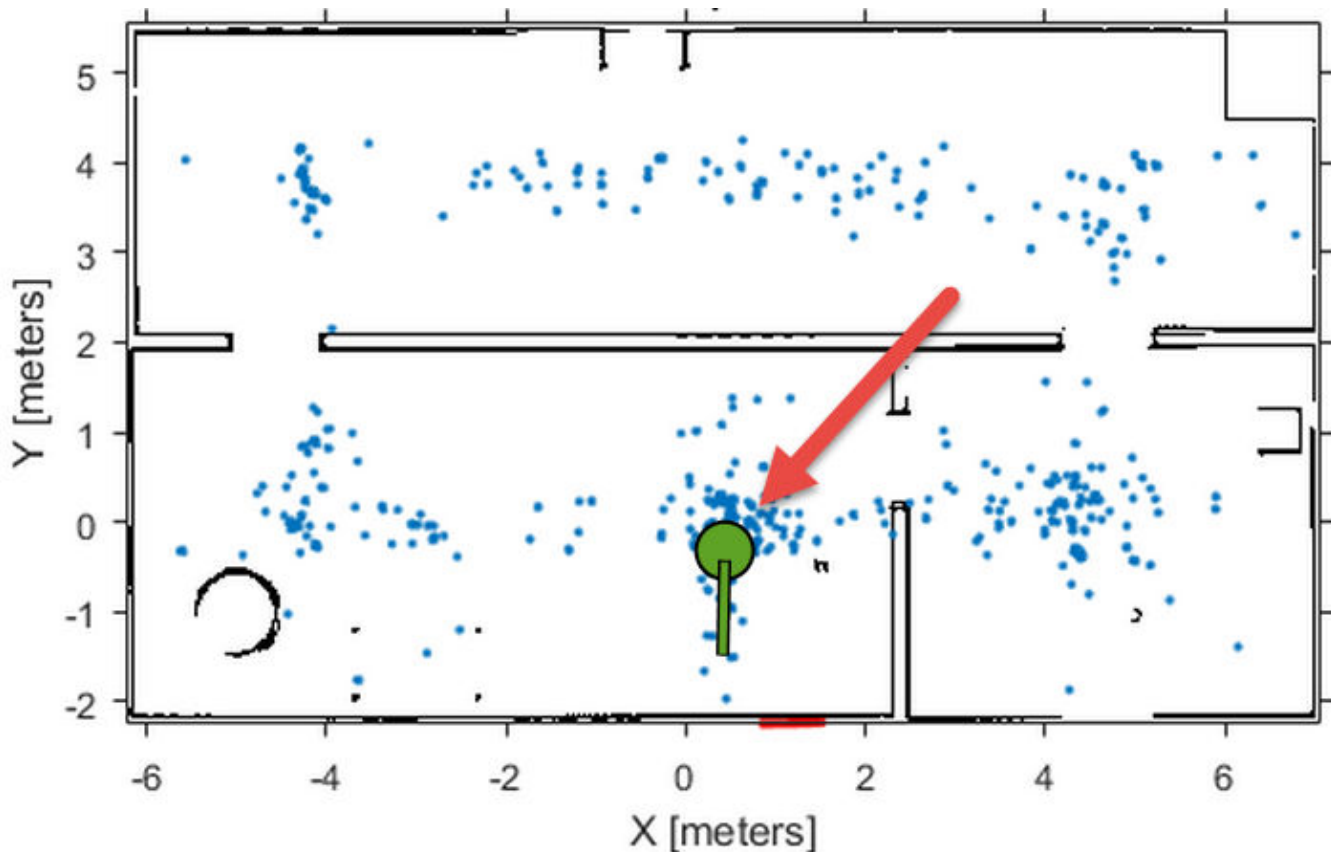
For more information on particle filters as a general application, see "Particle Filter Workflow" on page 2-58.

State Representation

When working with a localization algorithm, the goal is to estimate the state of your system. For robotics applications, this estimated state is usually a robot pose. For the `monteCarloLocalization` object, you specify this pose as a three-element vector. The pose corresponds to an x-y position, $[x \ y]$, and an angular orientation, `theta`.



The MCL algorithm estimates these three values based on sensor inputs of the environment and a given motion model of your system. The output from using the `monteCarloLocalization` object includes the pose, which is the best estimated state of the $[x \ y \ \text{theta}]$ values. Particles are distributed around an initial pose, `InitialPose`, or sampled uniformly using global localization. The pose is computed as the mean of the highest weighted cluster of particles once these particles have been corrected based on measurements.



This plot shows the highest weighted cluster and the final robot pose displayed over the samples particles in green. With more iterations of the MCL algorithm and measurement corrections, the particles converge to the true location of the robot. However, it is possible that particle clusters can have high weights for false estimates and converge on the wrong location. If the wrong convergence occurs, resample the particles by resetting the MCL algorithm with an updated `InitialPose`.

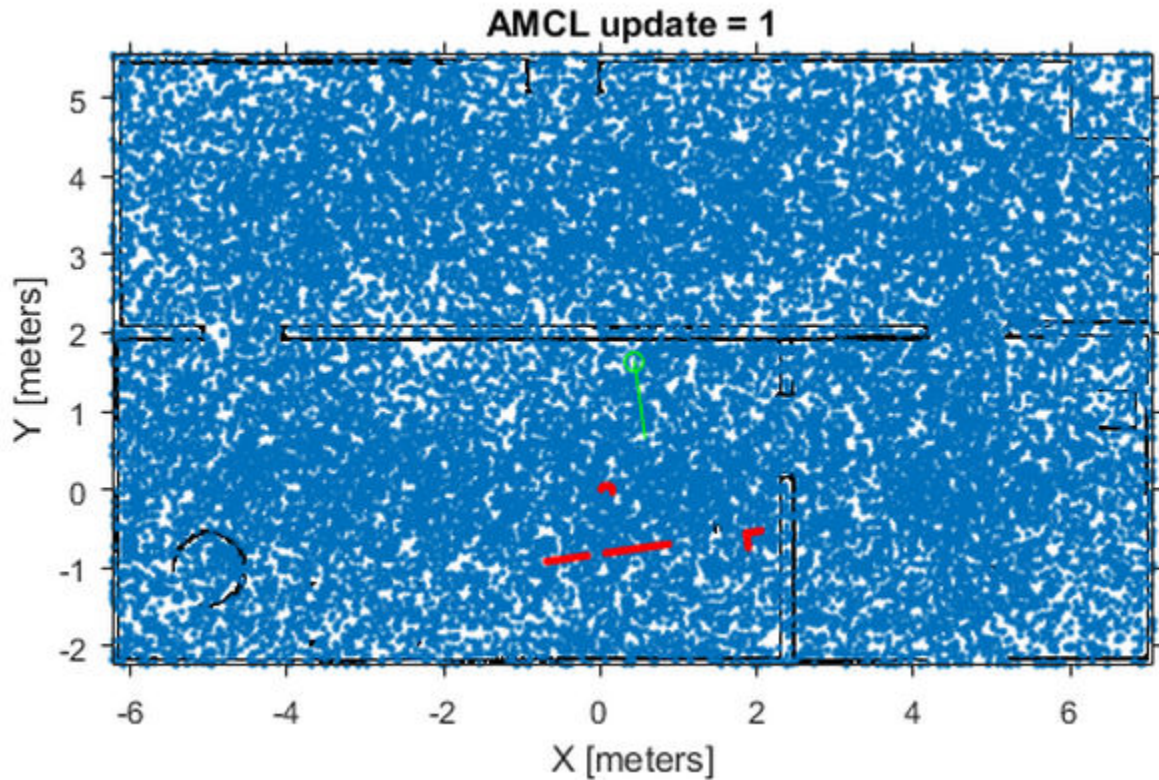
Initialization of Particles

When you first create the `monteCarloLocalization` algorithm, specify the minimum and maximum particle limits by using the `ParticleLimits` property. A higher number of particles increases the likelihood that the particles converge on the actual location. However, a lower particle number is faster. The number of particles adjusts dynamically within the limits based on the weights of particle clusters. This adjustment helps to reduce the number of particles over time so localization can run more efficiently.

Particle Distribution

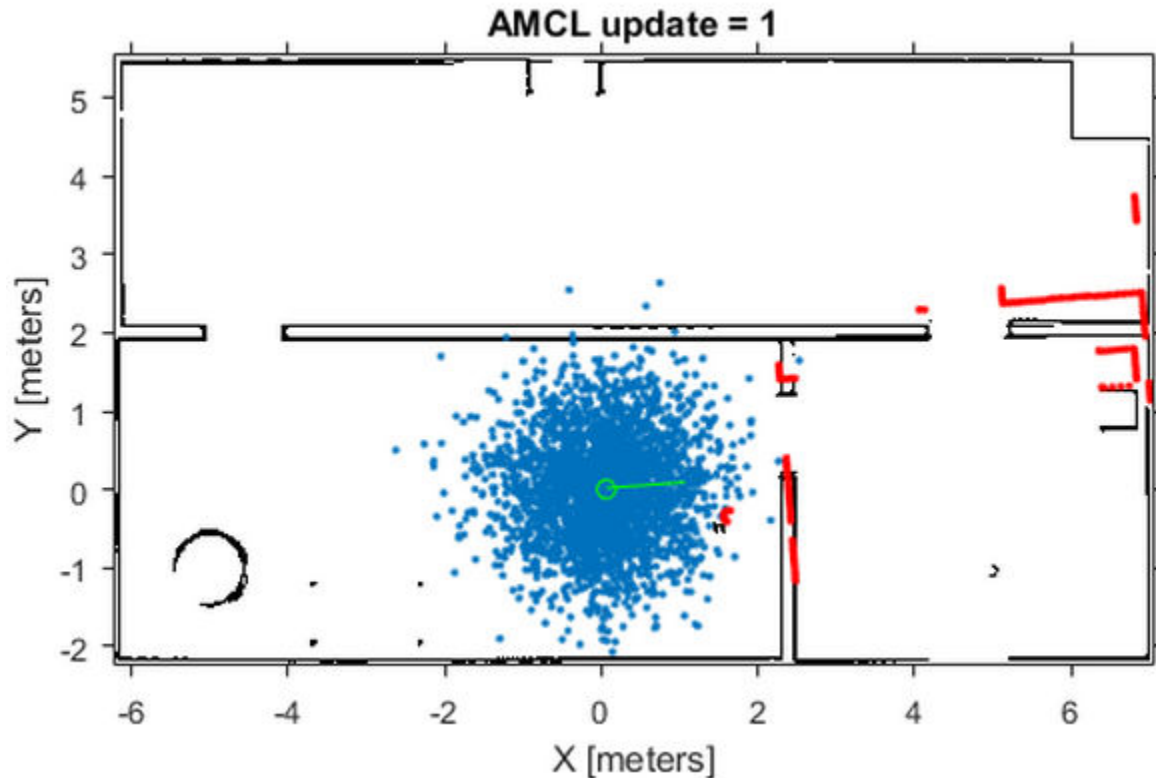
Particles must be sampled across a specified distribution. To initialize particles in the state space, you can use either an initial pose or global localization. With global localization, you can uniformly distribute particles across your expected state space (pulled from the `Map` property of your `SensorModel` object). In the default MCL object, set the `GlobalLocalization` property to `true`.

```
mcl = monteCarloLocalization;
mcl.GlobalLocalization = true;
```



Global localization requires a larger number of particles to effectively sample particles across the state space. More particles increase the likelihood of successful convergence on the actual state. This large distribution greatly reduces initial performance until particles begin to converge and particle number can be reduced.

By default, global localization is set to `false`. Without global localization, you must specify the `InitialPose` and `InitialCovariance` properties, which helps to localize the particles. Using this initial pose, particles are more closely grouped around an estimated state. A close grouping of particles enables you to use fewer of them, and increases the speed and accuracy of tracking during the first iterations.



These images were taken from the “Localize TurtleBot Using Monte Carlo Localization” on page 1-248 example, which shows how to use the MCL algorithm with the TurtleBot® in a known environment.

Resampling Particles and Updating Pose

To localize your robot continuously, you must resample the particles and update the algorithm. Use the `UpdateThreshold` and `ResamplingInterval` properties to control when resampling and updates to the estimated state occur.

The `UpdateThreshold` is a three-element vector that defines the minimum change in the robot pose, $[x \ y \ \theta]$, to trigger an update. Changing a variable by more than this minimum triggers an update, causing the object to return a new state estimate. This change in robot pose is based on the odometry, which is specified in the functional form of the object. Tune these thresholds based on your sensor properties and the motion of your robot. Random noise or minor variations greater than your threshold can trigger an unnecessary update and affect your performance. The `ResamplingInterval` property defines the number of updates to trigger particle resampling. For example, a resampling interval of 2 resamples at every other update.

The benefit of resampling particles is that you update the possible locations that contribute to the final estimate. Resampling redistributes the particles based on their weights and evolves particles based on the “Motion Model” on page 2-75. In this process, the particles with lower weight are eliminated, helping the particles converge to the true state of the robot. The number of particles dynamically changes to improve speed or tracking.

The performance of the algorithm depends on proper resampling. If particles are widely dispersed and the initial pose of the robot is not known, the algorithm maintains a high particle count. As the

algorithm converges on the true location, it reduces the number of particles and increases the speed of performance. You can tune your `ParticleLimits` property to limit the minimum and maximum particles used to help with the performance.

Motion and Sensor Model

The motion and sensor models for the MCL algorithm are similar to the `StateTransitionFcn` and `MeasurementLikelihoodFcn` functions for the `stateEstimatorPF` object, which are described in “Particle Filter Parameters” on page 2-62. For the MCL algorithm, these models are more specific to robot localization. After calling the object, to change the `MotionModel` or `SensorModel` properties, you must first call `release` on your object.

Sensor Model

By default, the `monteCarloLocalization` uses a `likelihoodFieldSensorModel` object as the sensor model. This sensor model contains parameters specific to the range sensor used, 2-D map information for the robot environment, and measurement noise characteristics. The sensor model uses the parameters with range measurements to compute the likelihood of the measurements given the current position of the robot. Without factoring in these parameters, some measurement errors can skew the state estimate or increase weight on irrelevant particles.

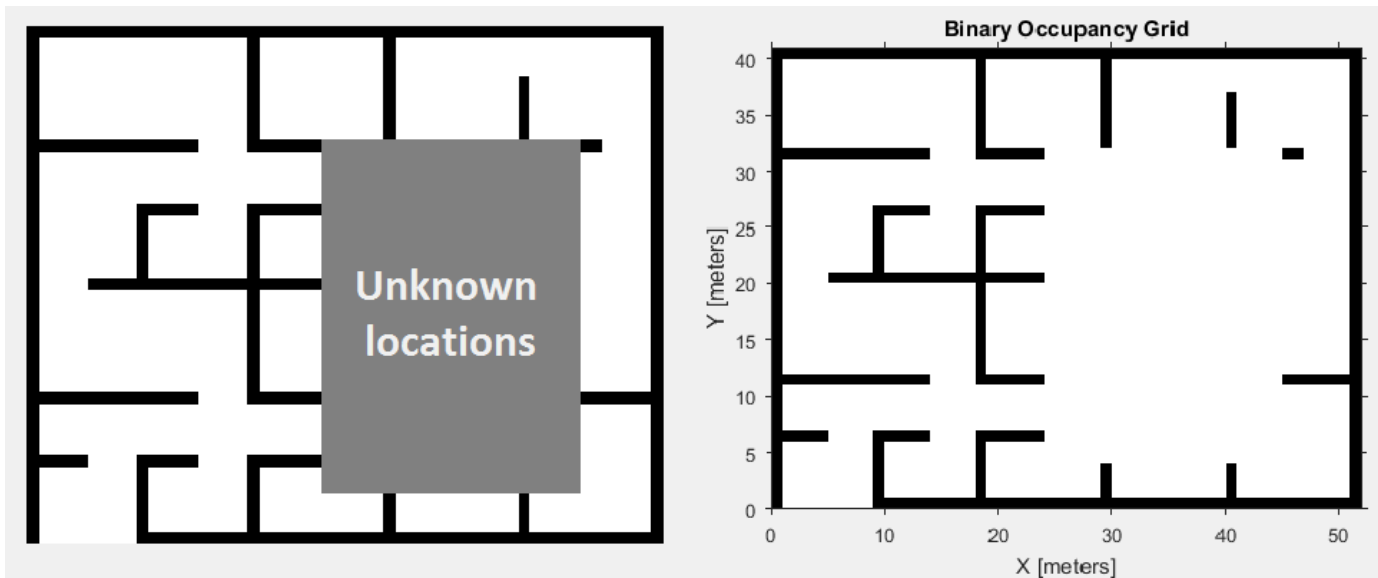
The range sensor properties are:

- `SensorPose` - The pose of the range sensor relative to the robot location. This pose is used to transform the range readings into the robot coordinate frame.
- `SensorLimits` - The minimum and maximum range limits. Measurement outside of these ranges are not factored into the likelihood calculation.
- `NumBeams` - Number of beams used to calculate likelihood. You can improve performance speed by reducing the number of beams used.

Range measurements are also known to give false readings due to system noise or other environmental interference. To account for the sensor error, specify these parameters:

- `MeasurementNoise` - Standard deviation for measurement noise. This deviation applies to the range reading and accounts for any interference with the sensor. Set this value based on information from your range sensor.
- `RandomMeasurementWeight` - Weight for probability of random measurement. Set a low probability for random measurements. The default is 0.05.
- `ExpectedMeasurementWeight` - Weight for probability of expected measurement. Set a high probability for expected measurements. The default is 0.95.

The sensor model also stores a map of the robot environment as an occupancy grid. Use `binaryOccupancyMap` to specify your map with occupied and free spaces. Set any unknown spaces in the map as free locations. Setting them to free locations prevents the algorithm from matching detected objects to these areas of the map.



Also, you can specify `MaximumLikelihoodDistance`, which limits the area for searching for obstacles. The value of `MaximumLikelihoodDistance` is the maximum distance to the nearest obstacle that is used for likelihood computation.

Motion Model

The motion model for robot localization helps to predict how particles evolve throughout time when resampling. It is a representation of robot kinematics. The motion model included by default with the MCL algorithm is an odometry-based differential drive motion model (`odometryMotionModel`). Without a motion model, predicting the next step is more difficult. It is important to know the capabilities of your system so that the localization algorithm can plan particle distributions to get better state estimates. Be sure to consider errors from the wheel encoders or other sensors used to measure the odometry. The errors in the system define the spread of the particle distribution.

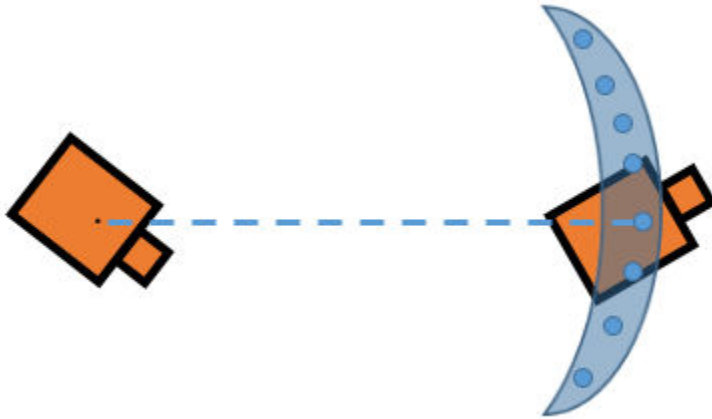
You can specify the error expected based on the motion of your robot as a four-element vector, `Noise`. These four elements are specified as weights on the standard deviations for [1]:

- Rotational error due to rotational motion
- Rotational error due to translational motion
- Translational error due to translational motion
- Translational error due to rotational motion

For differential drive robots, when a robot moves from a starting pose to a final pose, the change in pose can be treated as:

- 1 Rotation to the final position
- 2 Translation in a direct line to the final position
- 3 Rotation to the goal orientation

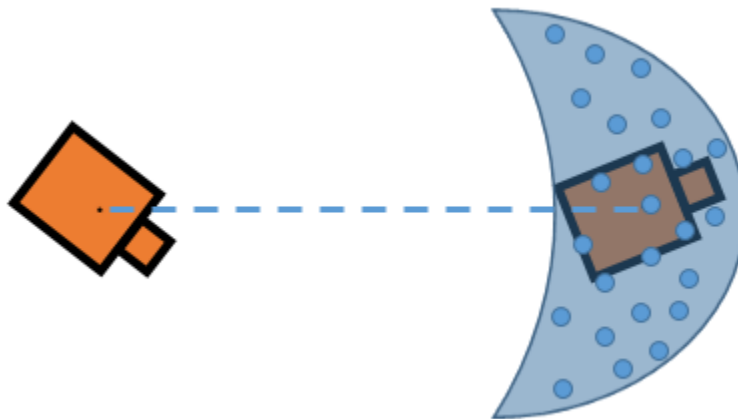
Assuming these steps, you can visualize the effect of errors in rotation and translation. Errors in the initial rotation result in your possible positions being spread out in a C-shape around the final position.



Large translational errors result in your possible positions being spread out around the direct line to the final position.



Large errors in both translation and rotation can result in wider-spread positions.



Also, rotational errors affect the orientation of the final pose. Understanding these effects helps you to define the Gaussian noise in the `Noise` property of the `MotionModel` object for your specific application. As the images show, each parameter does not directly control the dispersion and can vary with your robot configuration and geometry. Also, multiple pose changes as the robot navigates through your environment can increase the effects of these errors over many different steps. By accurately defining these parameters, particles are distributed appropriately to give the MCL algorithm enough hypotheses to find the best estimate for the robot location.

References

[1] Thrun, Sebastian, and Dieter Fox. *Probabilistic Robotics*. 3rd ed. Cambridge, Mass: MIT Press, 2006. p.136.

See Also

`monteCarloLocalization` | `likelihoodFieldSensorModel` | `odometryMotionModel`

Related Examples

- “Localize TurtleBot Using Monte Carlo Localization” on page 1-248

Vector Field Histogram

In this section...

“Robot Dimensions” on page 2-78

“Cost Function Weights” on page 2-80

“Histogram Properties” on page 2-80

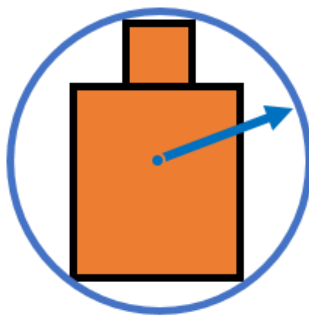
“Tune Parameters Using show” on page 2-82

The vector field histogram (VFH) algorithm computes obstacle-free steering directions for a robot based on range sensor readings. Range sensor readings are used to compute polar density histograms to identify obstacle location and proximity. Based on the specified parameters and thresholds, these histograms are converted to binary histograms to indicate valid steering directions for the robot. The VFH algorithm factors in robot size and turning radius to output a steering direction for the robot to avoid obstacles and follow a target direction.

Robot Dimensions

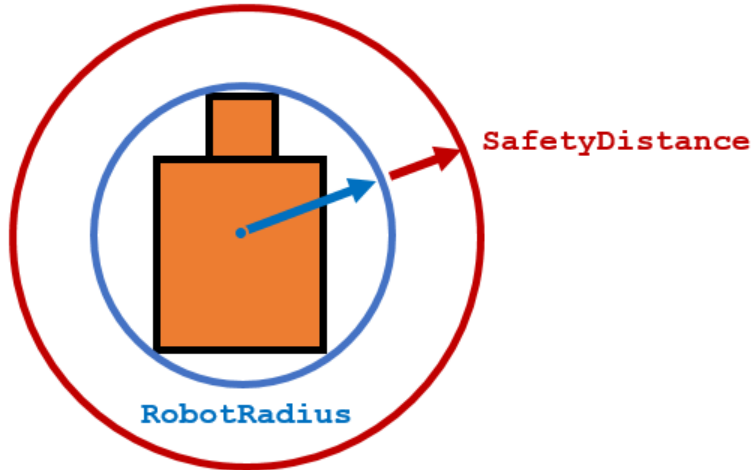
To calculate steering directions, you must specify information about the robot size and its driving capabilities. The VFH algorithm requires only four input parameters for the robot. These parameters are properties of the `controllerVFH` object: `RobotRadius`, `SafetyDistance`, `MinTurningRadius`, and `DistanceLimits`.

- `RobotRadius` specifies the radius of the smallest circle that can encircle all parts of the robot. This radius ensures that the robot avoids obstacles based on its size.

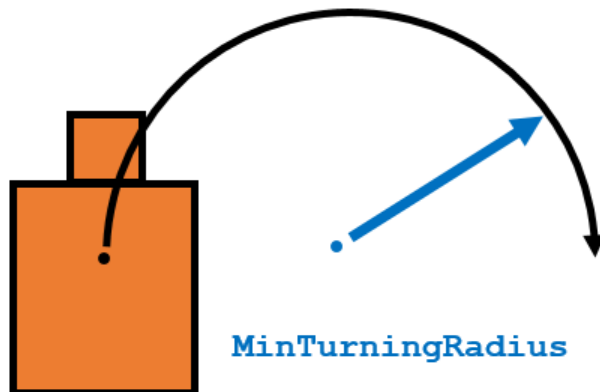


RobotRadius

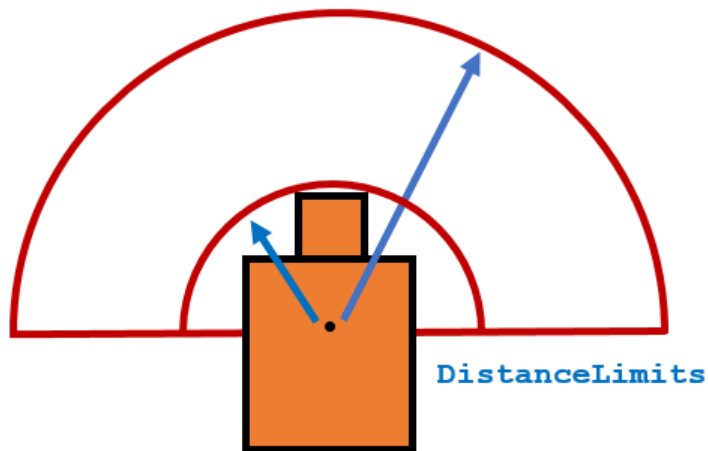
- `SafetyDistance` optionally specifies an added distance on top of the `RobotRadius`. You can use this property to add a factor of safety when navigating an environment.



- `MinTurningRadius` specifies the minimum turning radius for the robot traveling at the desired velocity. The robot may not be able to make sharp turns at high velocities. This property factors in navigating around obstacles and gives it enough space to maneuver.



- `DistanceLimits` specifies the distance range that you want to consider for obstacle avoidance. You specify the limits in a two-element vector, `[lower upper]`. The `lower` limit is used to ignore sensor readings that intersect with parts on the robot, sensor inaccuracies at short distances, or sensor noise. The `upper` limit is the effective range of the sensor or is based on your application. You might not want to consider all obstacles in the full sensor range.



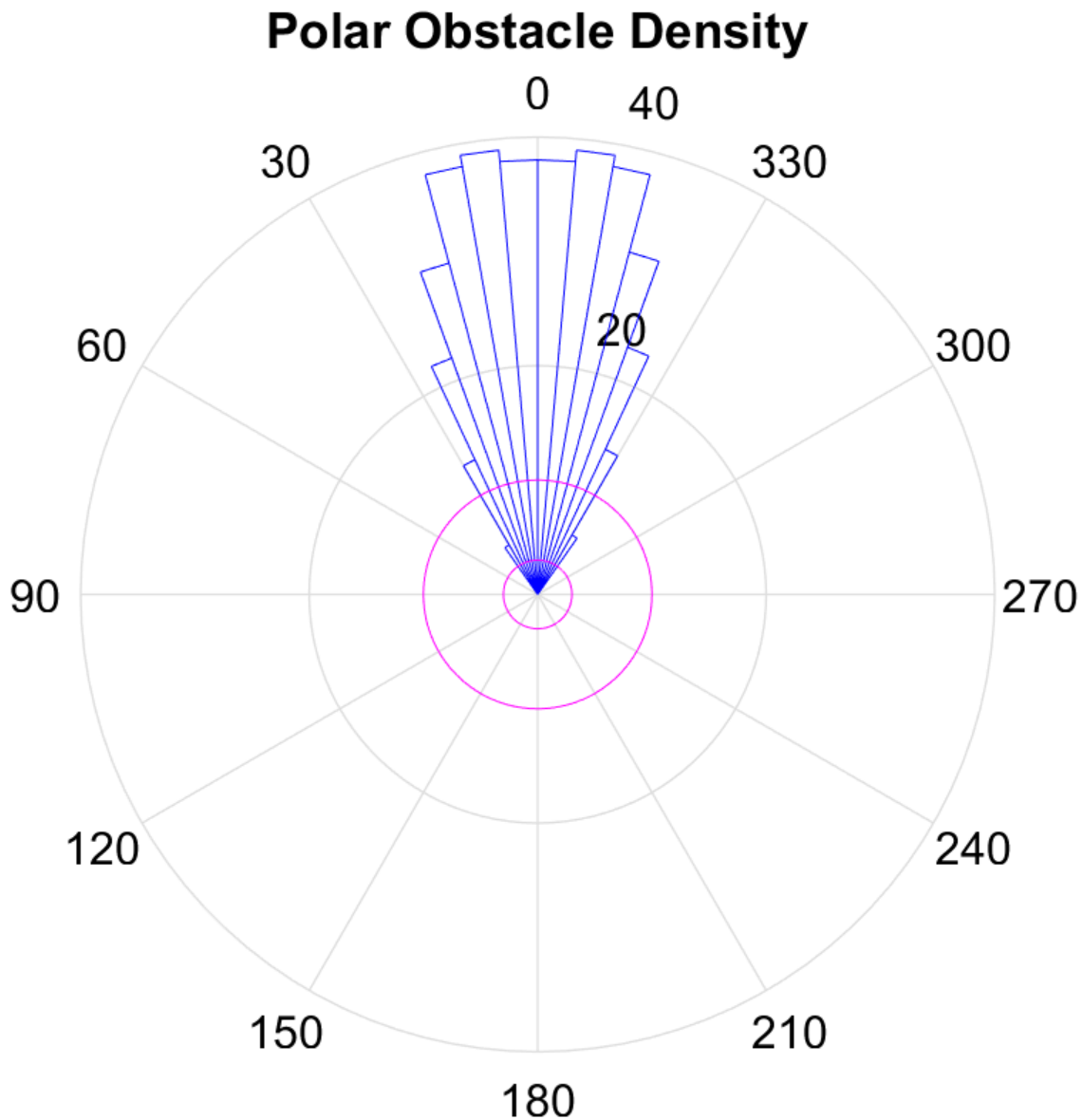
Note All information about the range sensor readings assumes that your range finder is mounted in the center of your robot. If the range sensor is mounted elsewhere, transform your range sensor readings from the laser coordinate frame to the robot base frame.

Cost Function Weights

Cost function weights are used to calculate the final steering directions. The VFH algorithm considers multiple steering directions based on your current, previous, and target directions. By setting the `CurrentDirectionWeight`, `PreviousDirectionWeight`, and `TargetDirectionWeight` properties, you can modify the steering behavior of your robot. Changing these weights affects the responsiveness of the robot and how it reacts to obstacles. To make the robot head towards its goal location, set `TargetDirectionWeight` higher than the sum of the other weights. This high `TargetDirectionWeight` value helps to ensure the computed steering direction is close to the target direction. Depending on your application, you might need to tune these weights.

Histogram Properties

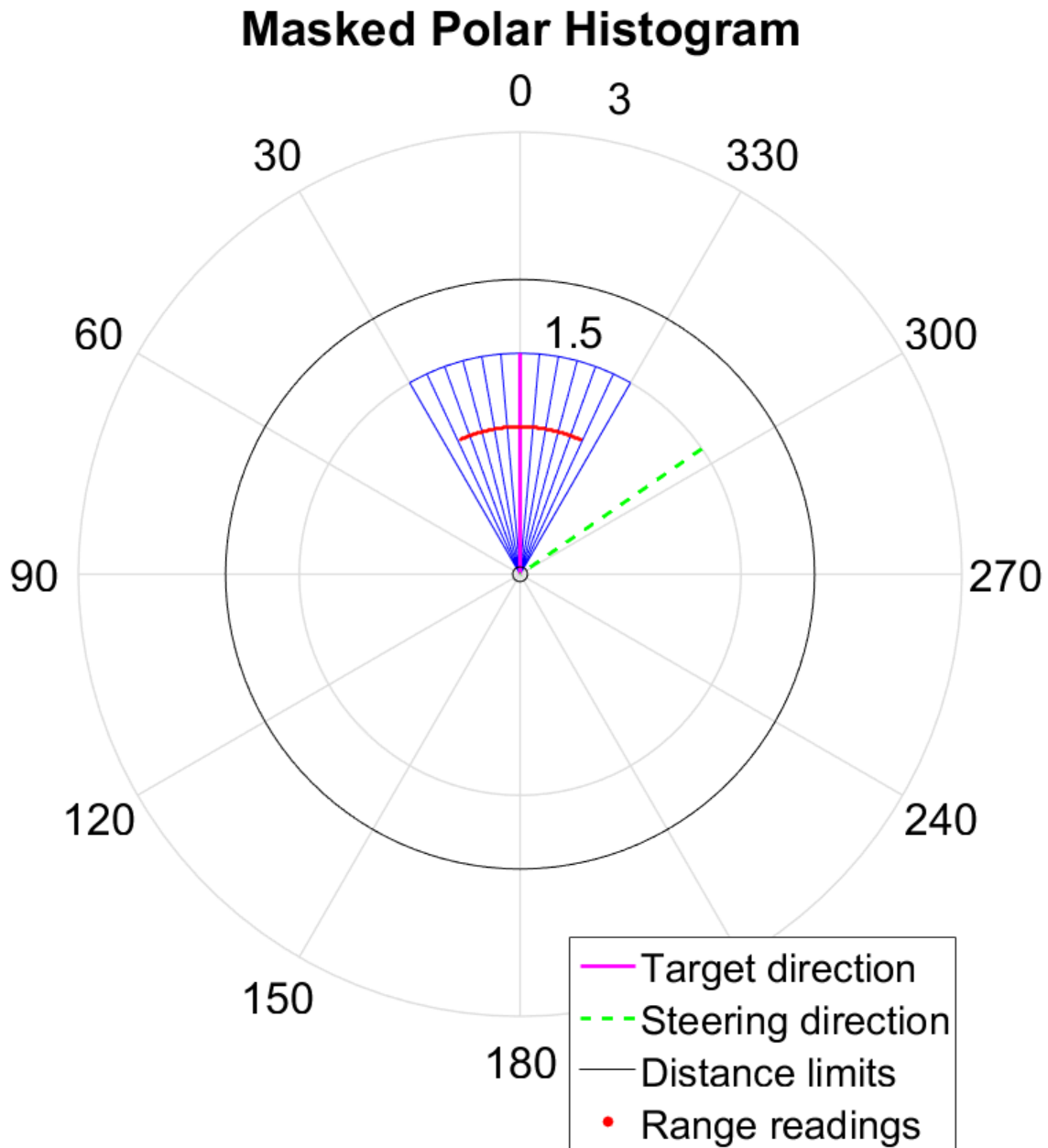
The VFH algorithm calculates a histogram based on the given range sensor data. It takes all directions around the robot and converts them to angular sectors that are specified by the `NumAngularSectors` property. This property is non-tunable and remains fixed once the `controllerVFH` object is called. The range sensor data is used to calculate a polar density histogram over these angular sectors.



Note Using a small NumAngularSectors value can cause the VFH algorithm to miss smaller obstacles. Missed obstacles do not appear on the histogram.

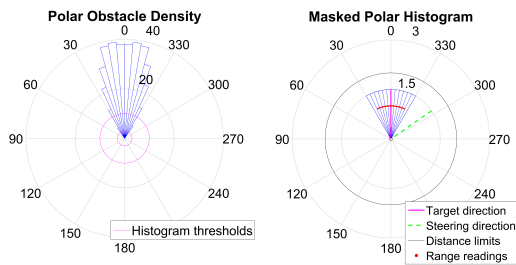
This histogram displays the angular sectors in blue and the histogram thresholds in pink. The HistogramThresholds property is a two-element vector that determines the values of the masked histogram, specified as [lower upper]. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the masked histogram. Values smaller than the lower threshold are represented as free space (0). Values that fall between the limits are set to the values in the previous binary histogram, with the default being free space (0). The masked histogram also factors in the MinTurningRadius, RobotSize, and SafetyDistance.

The polar density plot has the following corresponding masked histogram plot. This plot shows the target and steering directions, range readings, and distance limits.



Tune Parameters Using show

When working with a `controllerVFH` object, you can visualize the properties and parameters of the algorithm using the `show` function. This method displays the polar density plot and masked binary histogram. It also displays the algorithm parameters and the output steering direction for the VFH.



You can then tune parameters to help you prototype your obstacle avoidance application. For example, if you see that certain obstacles do not appear in the **Masked Polar Histogram** plot (right), then in the **Polar Obstacle Density** plot, consider adjusting the histogram thresholds to appropriate values. After you make the adjustments in the **Masked Polar Histogram** plot, the range sensor readings, shown in red, should match up with locations in the masked histogram (blue). Also, you can see the target and steering directions. You specify the target direction. The steering direction is the main output from the VFH algorithm. Adjusting the “Cost Function Weights” on page 2-80 can help you tune the output of the final steering direction.

Although you can use the show method in a loop, it slows computation speed due to the graphical plotting. If you are running this algorithm for real-time applications, get and display the VFH data in separate operations.

See Also
controllerVFH

Navigation Block Examples

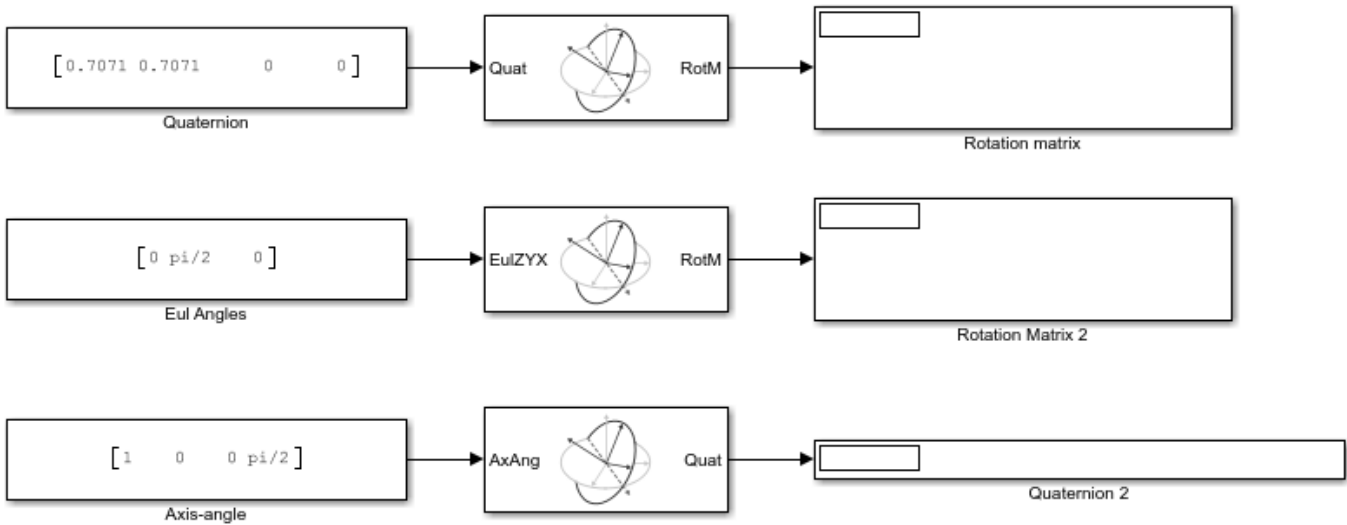
Convert Coordinate System Transformations

This model shows how to convert some basic coordinate system transformations into other coordinate systems. Input vectors are expected to be vertical vectors.

```
open_system('coord_trans_block_example_model.slx')
```

Warning: Unrecognized function or variable 'registerTICCS'.

Warning: Unrecognized function or variable 'customizationticcs'.



nmeaParser Examples

Plot Position of GNSS Receiver Using Live NMEA Data or NMEA Log File

This example shows how to parse information from NMEA sentences and use the obtained information to plot the location. In this example, you use the `nmeaParser` system object available in Navigation Toolbox to:

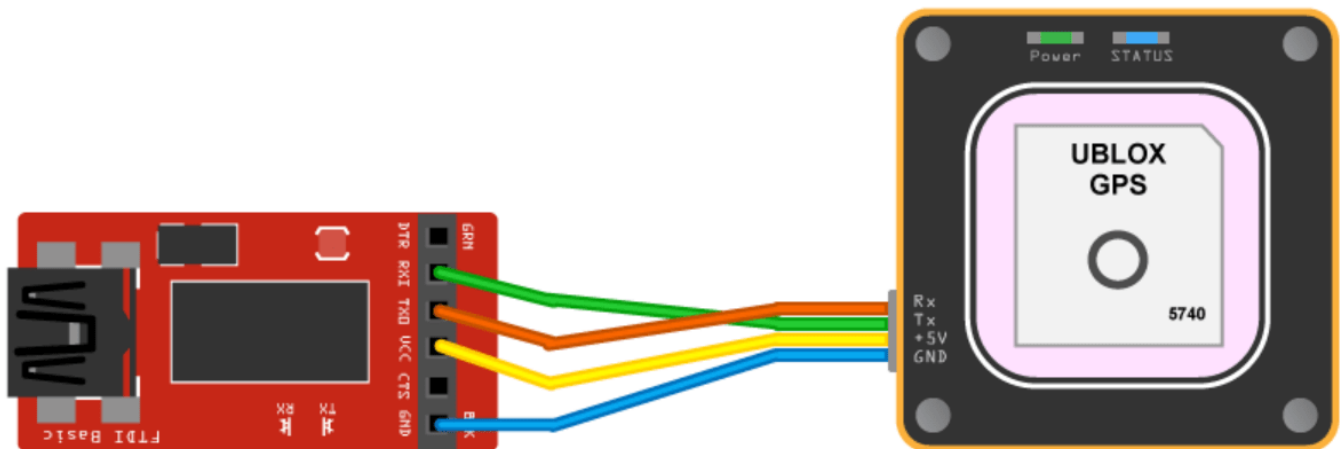
- Get the latitude and longitude from the live data obtained from a GNSS receiver and plot the location in a map.
- Get the latitude and longitude from the NMEA sentences stored in a log file and plot the location in a map.
- Get the position information of the satellites in view and plot the satellite azimuth and elevation data using `skyplot`.

Plot Location Using Live NMEA Data from a GNSS Receiver

You can read the location given by GNSS receiver connected to the host PC and plot the live location data. The latitude and longitude used to plot the location can be obtained from multiple NMEA sentences. In this section, we are using the RMC sentences obtained from the receiver to extract latitude and longitude.

Required Hardware

- Ublox NEO-6M GPS Module
- Serial-to-USB signal converters like FTDI (if the GPS module cannot be directly connected to computer using USB)



Hardware Setup

If a USB connector is available on the receiver, use a USB cable to connect the module directly to the host computer with Navigation Toolbox installed, and install the required drivers for the module. Otherwise, use serial-to-USB converters, to convert the serial transmissions to USB signals.

GNSS receivers require satellite signals to determine position information. The signals are acquired easily in locations that have a clear view of the sky. Ensure that you keep the module or receiver

antenna in such a way so that it gets a clear view to the sky. The receiver might take some time to acquire signals.

Connect to Receiver and Parse NMEA Data

Create a connection to the serial port where the GNSS receiver is connected by specifying the port and baudrate.

```
% Create a serial object.
port = 'com8';
baudrate = 9600;
gpsObj = serialport(port,baudrate);
```

Create a nmeaParser object by specifying the Message ID of the sentence to be parsed as "RMC"

```
parserObj = nmeaParser("MessageId","RMC");
```

Read the values obtained from the receiver, parse the value to get the latitude and longitude, and plot the location on a map along with the timestamp, for a duration specified by timeOut.

```
an = [];
timeOut = 10;
ts = tic;
while(toc(ts)<timeOut)
    % Read line by line from the serial object.
    data = readline(gpsObj);
    rmcData = parserObj(data);
    % Status = 0, indicates, the input NMEA sentence is an valid RMC
    % sentence.
    if rmcData.Status == 0
        fixStatus = rmcData.FixStatus;
        latitude = rmcData.Latitude;
        longitude = rmcData.Longitude;
        gpsTime = rmcData.UTCDateTime;
        % Plot the position in geographic coordinates.
        geoplot(latitude,longitude,'Marker',"diamond",'MarkerSize',10,'Color','b', ...
            'MarkerFaceColor','r');
        % Selects the basemap.
        geobasemap streets;
        % Fix Status A represents the satellite fix is obtained.
        if fixStatus == "A"
            % Adjust the geographic limits of the current geographic axes.
            geolimits([latitude-2.5,latitude+2.5],[longitude-2.5,longitude+2.5]) ;
            txt = strcat("Timestamp: ",string(gpsTime));
        else
            txt = "No Fix";
        end
        % Update time or Fix Status on the figure.
        delete(an);
        an = annotation('textbox', [0.005, 0.98, 0.6, 0.01], 'FitBoxToText','on', ...
            'string', txt,'Color','blue','FontSize',10);
    end
end
```



Plot Location by Reading NMEA Sentences Stored in a Log file

This section shows how to read GNSS receiver data in NMEA format logged in a text file, parse the raw data to get the location information, and plot the location obtained in the map. The latitude and longitude used to plot the location can be obtained from multiple NMEA sentences. In this section, we are using GGA sentence to extract latitude and longitude.

Open the log file included with this example and read the data from the file.

```
% Opens the file gpsLog.txt with read access.
fileID = fopen('gpsLog.txt','r');
% Read the text file.
gpsData = fscanf(fileID,'%c');
```

Create nmeaParser object by specifying the message ID as GGA. Use the nmeaParser object to parse the data read from the log file.

```
parserObj = nmeaParser('MessageId','GGA');
% Parse the NMEA Data.
ggaData = parserObj(gpsData);
```

Read the location from the parsed value and plot the location in the map to see the path travelled by the GNSS receiver.

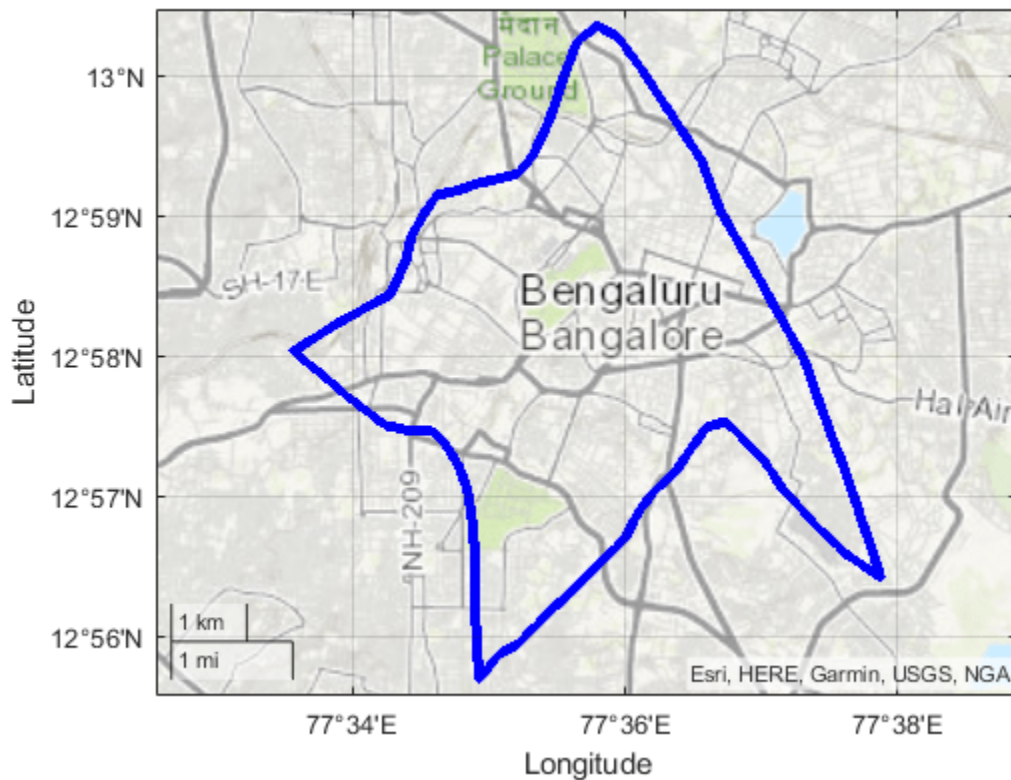
```
% Initialize variables.
latVector = zeros(1,numel(ggaData));
lonVector = zeros(1,numel(ggaData));
```

```

for i=1:length(ggaData)
    % Check if the parsed GGA sentences are valid and if they are valid, get the
    % latitude and longitude from the output structures. Status = 0,
    % indicates the data is valid
    if ggaData(i).Status == 0
        latVector(i) = ggaData(i).Latitude;
        lonVector(i) = ggaData(i).Longitude;
    end
end
% Remove Nan value in latitude and longitude data, if any. nmeaParser object
% returns NaN for a value if the value is not available in the sentence.
% For example, latitude and longitude data are not available if there is no
% satellite fix.
latVector = latVector(~isnan(latVector));
lonVector = lonVector(~isnan(lonVector));

% Plot the position in geographic coordinates
geoplot(latVector,lonVector,'Marker','*','MarkerSize',3, ...
    'Color','blue','MarkerFaceColor','red');
% Selects the basemap
geobasemap 'topographic';

```



Plot Satellite Position Using skyplot

In this section, you read the satellite position information from the GSV sentences stored in an NMEA log file and plot the azimuth and elevation data using `skyplot`.

Open the log file included with this example and read the data from the file.

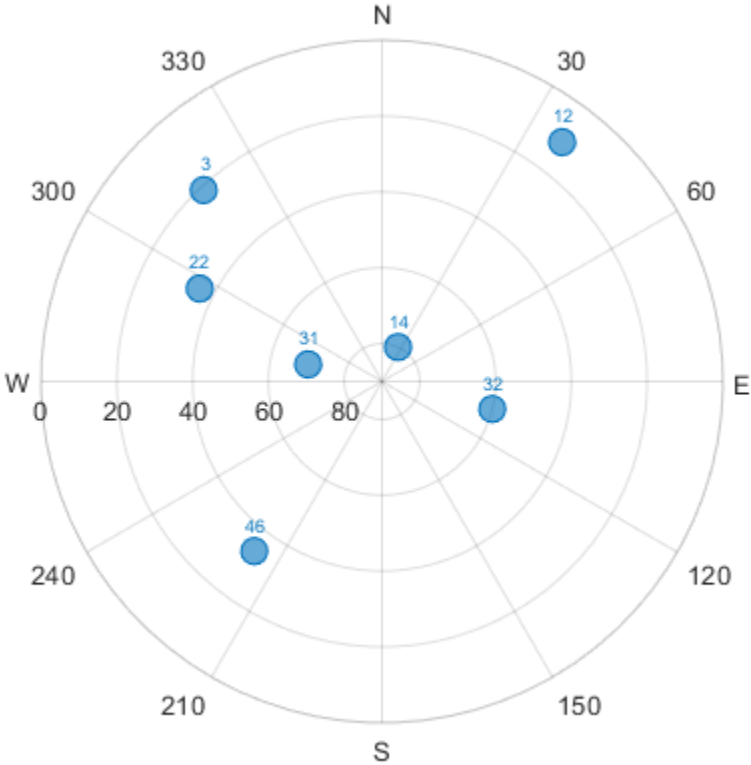
```
% Opens the file gpsLog.txt with read access
fileID = fopen('gpsLog2.txt','r');
% Read the text file
gpsData = fscanf(fileID,'%c');
```

Create nmeaParser object specifying the message ID as GSV. Use the nmeaParser object to parse the data read from the log file.

```
parserObj = nmeaParser('MessageId','GSV');
% Parse the NMEA Data
gsvData = parserObj(gpsData);
```

Read the azimuth and elevation from the parsed data and plot the satellite position using skyplot. The complete satellite information might be available in multiple GSV sentences. In this section, we are combining all satellite information per cycle before plotting them.

```
az = []; el = []; satID = []; prevSentenceNumber = 0;
% Create an empty skyplot
sp = skyplot([], [],[]);
for dataCount = 1:numel(gsvData)
    % The complete satellite information for the GPS frame might be
    % available in multiple GSV sentences. The various fields in the GSV
    % sentence can be used to combine the information into a single frame.
    % In this example, we use sentence number and number of satellites in
    % view to combine the information. The SentenceNumber of the GSV data
    % is expected to be sequential for a given frame of GPS data and total
    % number of azimuth and elevation data should be equal to satellites
    % in view.
    if gsvData(dataCount).SentenceNumber == prevSentenceNumber + 1
        az = [az, gsvData(dataCount).Azimuth];
        el = [el, gsvData(dataCount).Elevation];
        satID = [satID, gsvData(dataCount).SatelliteID];
        prevSentenceNumber = gsvData(dataCount).SentenceNumber;
        satellitesInView = gsvData(dataCount).SatellitesInView;
        % Once all the satellite information in a GPS frame is obtained,
        % plot the satellite position using skyplot
        if numel(az) == satellitesInView && numel(el) == satellitesInView
            set(sp, 'AzimuthData', az, 'ElevationData', el, 'LabelData', satID);
            drawnow;
            az = []; el = []; satID = []; prevSentenceNumber = 0;
            % The pause is just to see the satellite position changes in
            % the plot
            pause(2);
        end
    else
        az = []; el = []; satID = []; prevSentenceNumber = 0;
    end
end
end
```



See also

- `skyplot`
- `nmeaParser`

