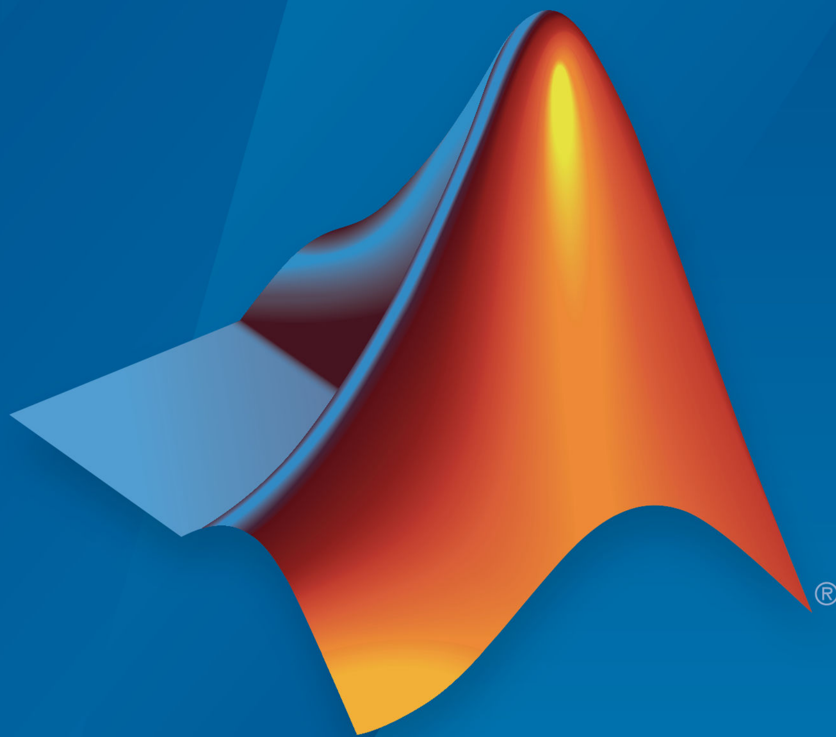
Navigation Toolbox™

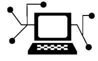
User's Guide



MATLAB® & SIMULINK®

R2019**b**

How to Contact MathWorks



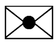
Latest news: www.mathworks.com

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 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)

Contents

1 Navigation Featured Examples

Estimate Position and Orientation of a Ground Vehicle 1-3

Pose Estimation From Asynchronous Sensors 1-13

Inertial Sensor Noise Analysis Using Allan Variance 1-20

Estimate Orientation Through Inertial Sensor Fusion 1-34

IMU and GPS Fusion for Inertial Navigation 1-46

Magnetometer Calibration 1-56

Remove Bias from Angular Velocity Measurement 1-67

Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter 1-72

Rotations, Orientation, and Quaternions 1-78

Lowpass Filter Orientation Using Quaternion SLERP 1-97

Introduction to Simulating IMU Measurements 1-103

Logged Sensor Data Alignment for Orientation Estimation 1-118

Estimate Robot Pose with Scan Matching 1-127

Localize TurtleBot Using Monte Carlo Localization 1-136

Compose a Series of Laser Scans with Pose Changes 1-152

Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs **1-157**

Create Egocentric Occupancy Maps Using Range Sensors . **1-161**

Build Occupancy Map from Lidar Scans and Poses **1-168**

Create Egocentric Occupancy Map from Driving Scenario Designer **1-170**

Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph **1-176**

Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans **1-181**

Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans **1-190**

Perform SLAM Using 3-D Lidar Point Clouds **1-198**

Plan Mobile Robot Paths using RRT **1-210**

Moving Furniture in a Cluttered Room with RRT **1-217**

Motion Planning with RRT for a Robot Manipulator **1-224**

Dynamic Replanning on an Indoor Map **1-232**

Lane Change for Highway Driving **1-239**

Path Following with Obstacle Avoidance in Simulink® **1-257**

Obstacle Avoidance with TurtleBot and VFH **1-263**

Optimal Trajectory Generation for Urban Driving **1-266**

2 Navigation Topics

Model IMU, GPS, and INS/GPS **2-2**
Inertial Measurement Unit **2-2**
Global Positioning System **2-5**
Inertial Navigation System and Global Positioning System . . . **2-9**

Occupancy Grids **2-10**
Overview **2-10**
World, Grid, and Local Coordinates **2-11**
Inflation of Coordinates **2-12**
Log-Odds Representation of Probability Values **2-18**

Execute Code at a Fixed-Rate **2-21**
Introduction **2-21**
Run Loop at Fixed Rate **2-21**
Overrun Actions for Fixed Rate Execution **2-22**

Particle Filter Workflow **2-24**
Estimation Workflow **2-24**

Particle Filter Parameters **2-29**
Number of Particles **2-29**
Initial Particle Location **2-30**
State Transition Function **2-32**
Measurement Likelihood Function **2-33**
Resampling Policy **2-33**
State Estimation Method **2-34**

Pure Pursuit Controller **2-36**
Reference Coordinate System **2-36**
Look Ahead Distance **2-37**
Limitations **2-38**

Monte Carlo Localization Algorithm **2-39**
Overview **2-39**
State Representation **2-40**
Initialization of Particles **2-42**
Resampling Particles and Updating Pose **2-44**
Motion and Sensor Model **2-45**

Vector Field Histogram **2-50**
Robot Dimensions **2-50**
Cost Function Weights **2-52**
Histogram Properties **2-53**
Tune Parameters Using show **2-57**

3 Navigation Block Examples

Convert Coordinate System Transformations **3-2**

1

Navigation Featured Examples

- “Estimate Position and Orientation of a Ground Vehicle” on page 1-3
- “Pose Estimation From Asynchronous Sensors” on page 1-13
- “Inertial Sensor Noise Analysis Using Allan Variance” on page 1-20
- “Estimate Orientation Through Inertial Sensor Fusion” on page 1-34
- “IMU and GPS Fusion for Inertial Navigation” on page 1-46
- “Magnetometer Calibration” on page 1-56
- “Remove Bias from Angular Velocity Measurement” on page 1-67
- “Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter” on page 1-72
- “Rotations, Orientation, and Quaternions” on page 1-78
- “Lowpass Filter Orientation Using Quaternion SLERP” on page 1-97
- “Introduction to Simulating IMU Measurements” on page 1-103
- “Logged Sensor Data Alignment for Orientation Estimation” on page 1-118
- “Estimate Robot Pose with Scan Matching” on page 1-127
- “Localize TurtleBot Using Monte Carlo Localization” on page 1-136
- “Compose a Series of Laser Scans with Pose Changes” on page 1-152
- “Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs” on page 1-157
- “Create Egocentric Occupancy Maps Using Range Sensors” on page 1-161
- “Build Occupancy Map from Lidar Scans and Poses” on page 1-168
- “Create Egocentric Occupancy Map from Driving Scenario Designer” on page 1-170
- “Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph” on page 1-176
- “Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans” on page 1-181
- “Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans” on page 1-190
- “Perform SLAM Using 3-D Lidar Point Clouds” on page 1-198

- “Plan Mobile Robot Paths using RRT” on page 1-210
- “Moving Furniture in a Cluttered Room with RRT” on page 1-217
- “Motion Planning with RRT for a Robot Manipulator” on page 1-224
- “Dynamic Replanning on an Indoor Map” on page 1-232
- “Lane Change for Highway Driving” on page 1-239
- “Path Following with Obstacle Avoidance in Simulink®” on page 1-257
- “Obstacle Avoidance with TurtleBot and VFH” on page 1-263
- “Optimal Trajectory Generation for Urban Driving” on page 1-266

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 that 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
```

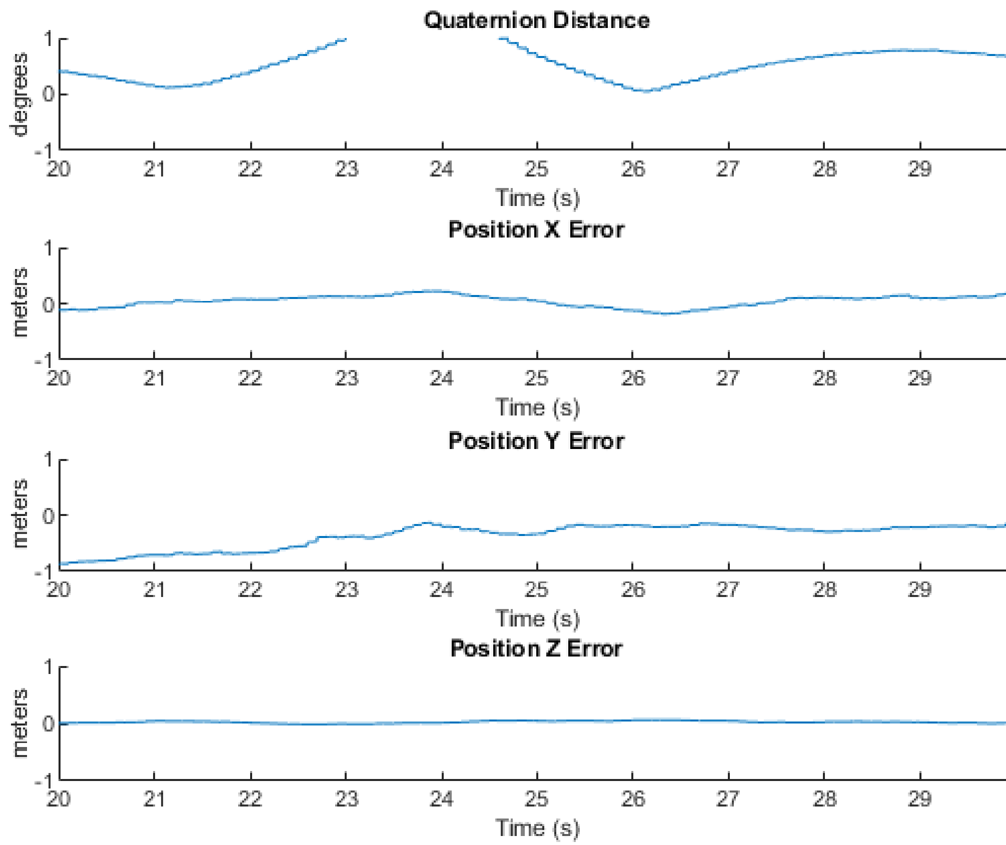
```
if usePoseView
    viewer = HelperPoseViewer( ...
        'XPositionLimits', [-15, 15], ...
        'YPositionLimits', [-15, 15], ...
        'ZPositionLimits', [-5, 5], ...
        'ReferenceFrame', 'ENU');
end
```

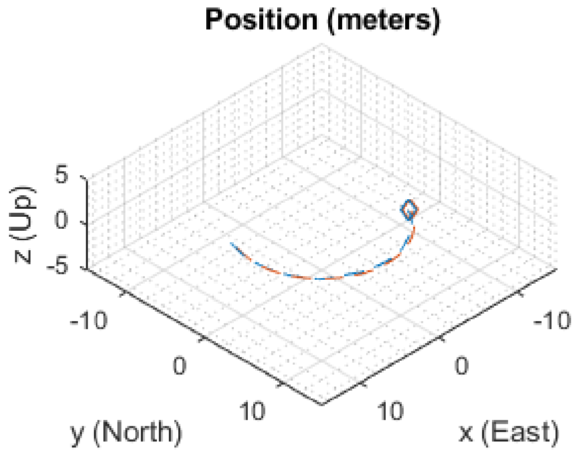
Simulation Loop

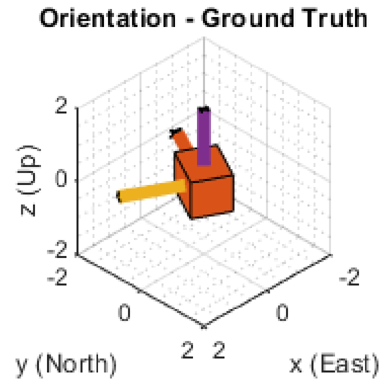
The main simulation loop is a while loop with a nested for loop. The while loop executes at the `gpsFs`, which is the GPS measurement rate. The nested for loop executes at the `imuFs`, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

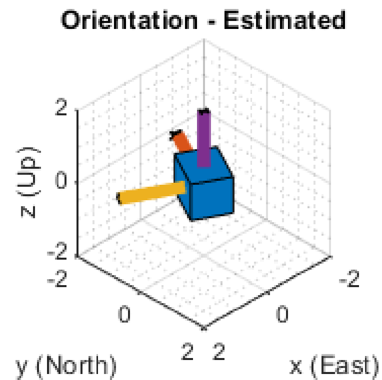
```
totalSimTime = 30; % seconds

% 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,:);
                    errscope(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

        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
```


Quaternion Distance
degrees
1
0
-1
20
21
22
23
24
25
26
27
28
29
Time (s)
Position X Error
meters
Time (s)
Position Y Error
meters
Time (s)
Position Z Error
meters
Time (s)







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.

quatd = rad2deg(dist(estOrientation, trueOrientation));
```

```
% Display RMS errors in the command window.
fprintf('\n\nEnd-to-End Simulation Position RMS Error\n');
msep = sqrt(mean(posd.^2));
fprintf('\tX: %.2f , Y: %.2f, Z: %.2f   (meters)\n\n', msep(1), ...
    msep(2), msep(3));

fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));
```

```
End-to-End Simulation Position RMS Error
    X: 1.16 , Y: 0.99, Z: 0.03   (meters)

End-to-End Quaternion Distance RMS Error (degrees)
    0.51 (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.

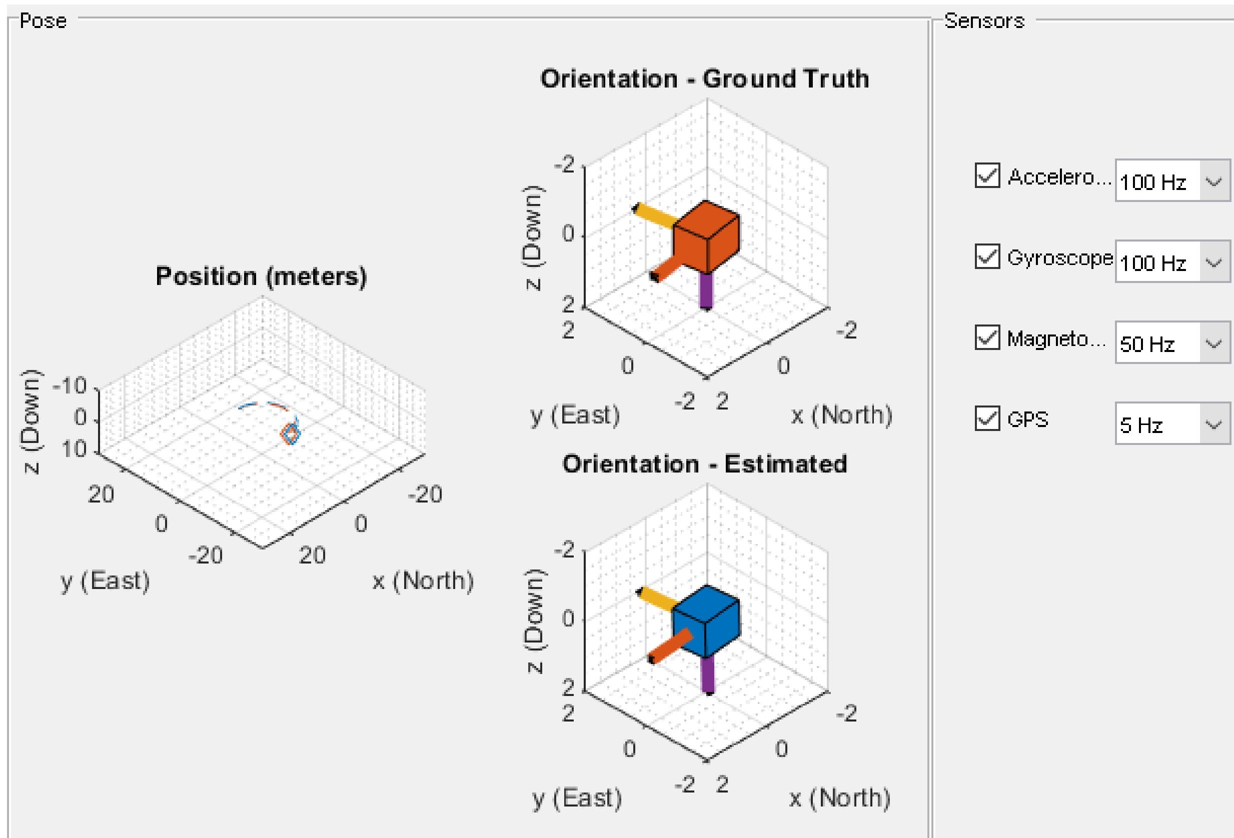
Some configurations produce dramatic results. For example, turning off the GPS sensor causes the position estimate to drift quickly. Turning off the magnetometer sensor will cause the orientation estimate to slowly deviate from the ground truth as the estimate rotates too fast. Conversely, if the gyroscope 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.

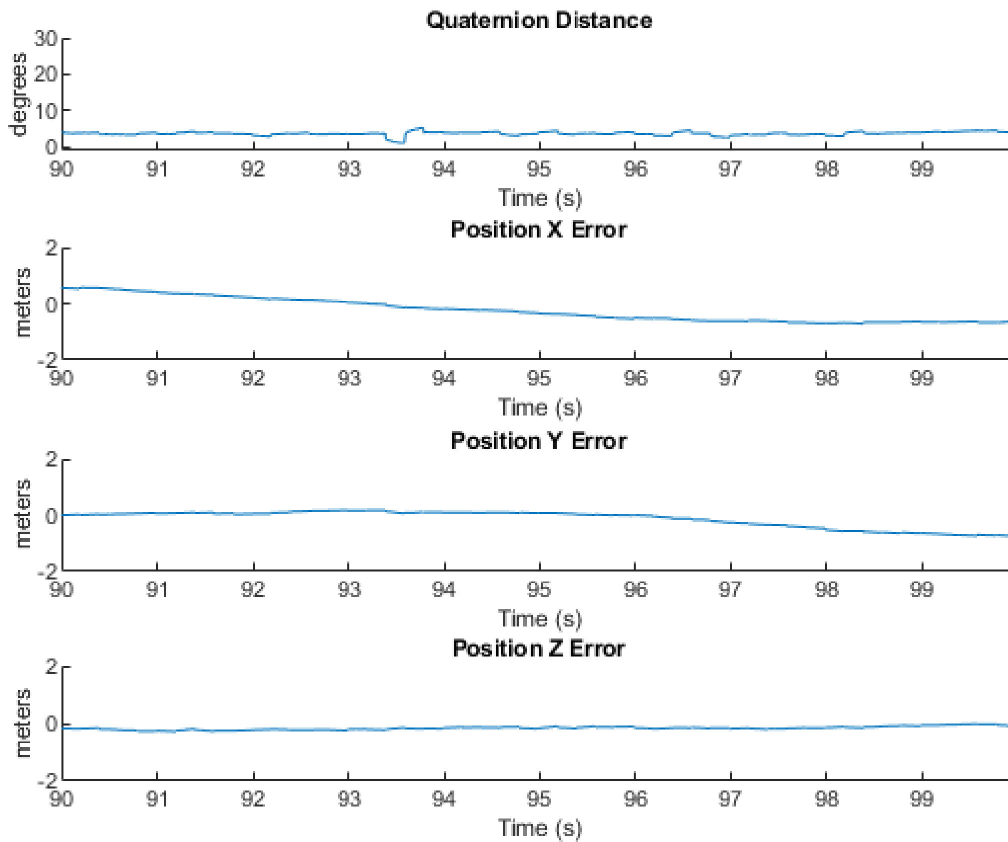
Turning all sensors on but setting them to run at the lowest rate produces an estimate that visibly deviates from the ground truth and then snaps back to a more correct result when sensors are fused. This is most easily seen in the `HelperScrollingPlotter` of the running estimate errors.

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,:), Racc);
    end

    % Fuse Gyroscope
    if (f.UserData.Gyroscope) && ...
```

```
        mod(ii, fix(Fs/f.UserData.GyroscopeSampleRate)) == 0

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


Pose
Sensors
Orientation - Ground Truth
Position (meters)
Orientation - Estimated
z (Down)
y (East)
x (North)
Accelero...
100 Hz
Gyroscope
100 Hz
Magneto...
50 Hz
GPS
5 Hz



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 τ_0, $2\tau_0$, ..., $m\tau_0, (m < (L-1)/2)$ 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:

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

where $\tau = m\tau_0$ and $<>$ is the ensemble average.

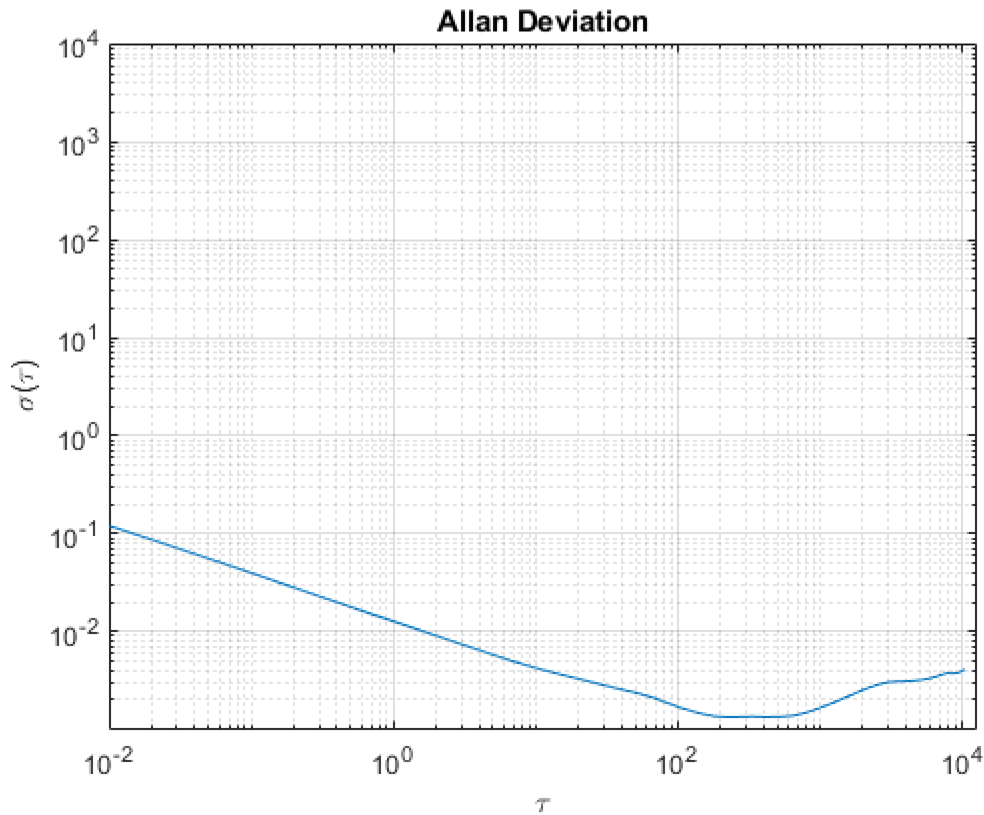
The ensemble average can be expanded to:

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

```
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));
```

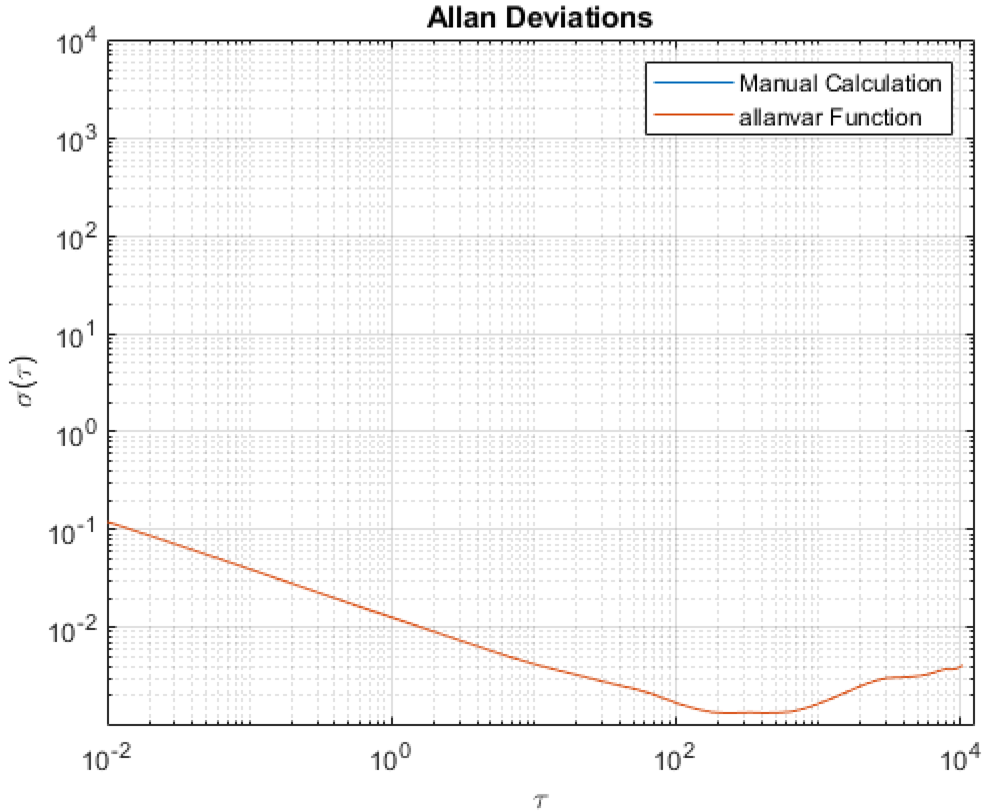
Finally, the Allan deviation $\sigma(t) = \sqrt{\sigma^2(t)}$ is used to determine the gyroscope noise parameters.

```
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:

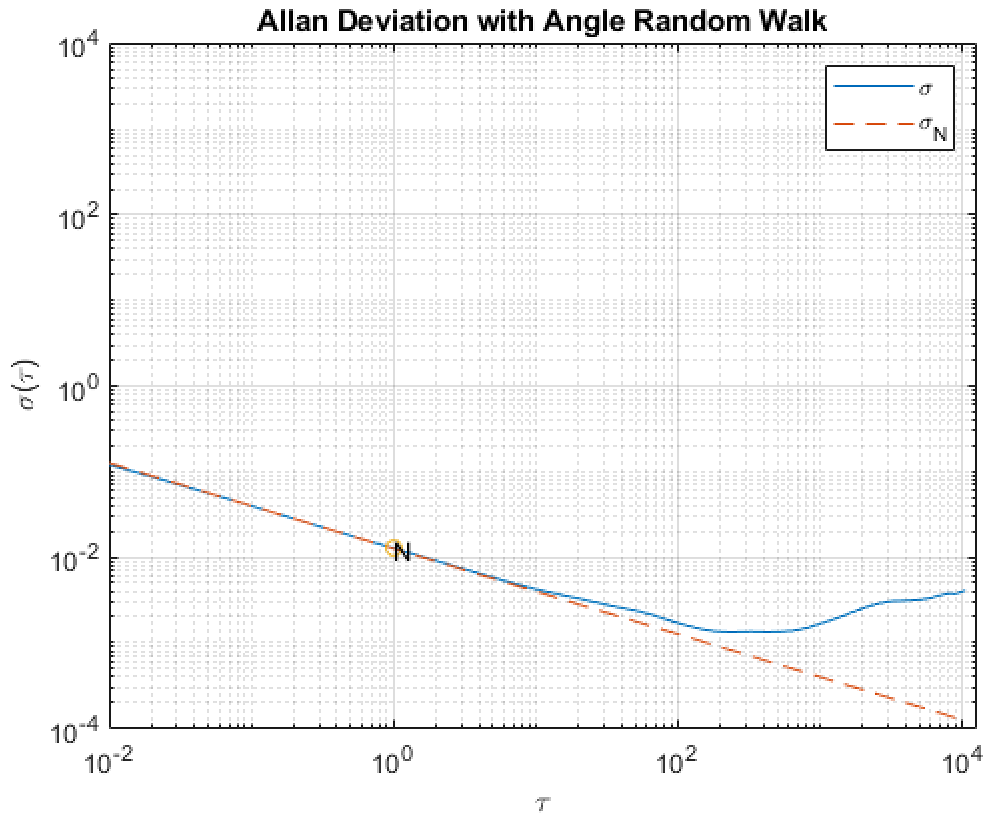
$$\sigma^2(\tau) = \frac{N^2}{\tau}$$

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 $\tau = 1$.

```
% 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:

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

where

K = rate random walk coefficient

Substituting into the original PSD equation and performing integration yields:

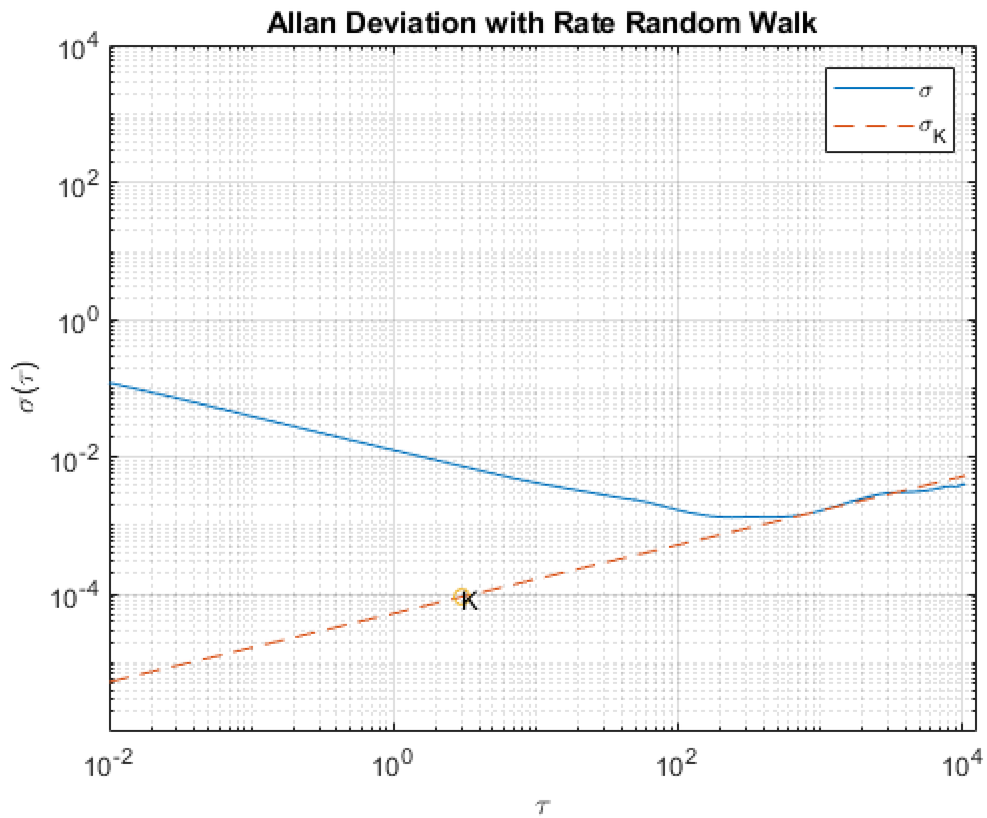
$$\sigma^2(\tau) = \frac{K^2\tau}{3}$$

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 $\tau = 3$.

```
% 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:

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

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:

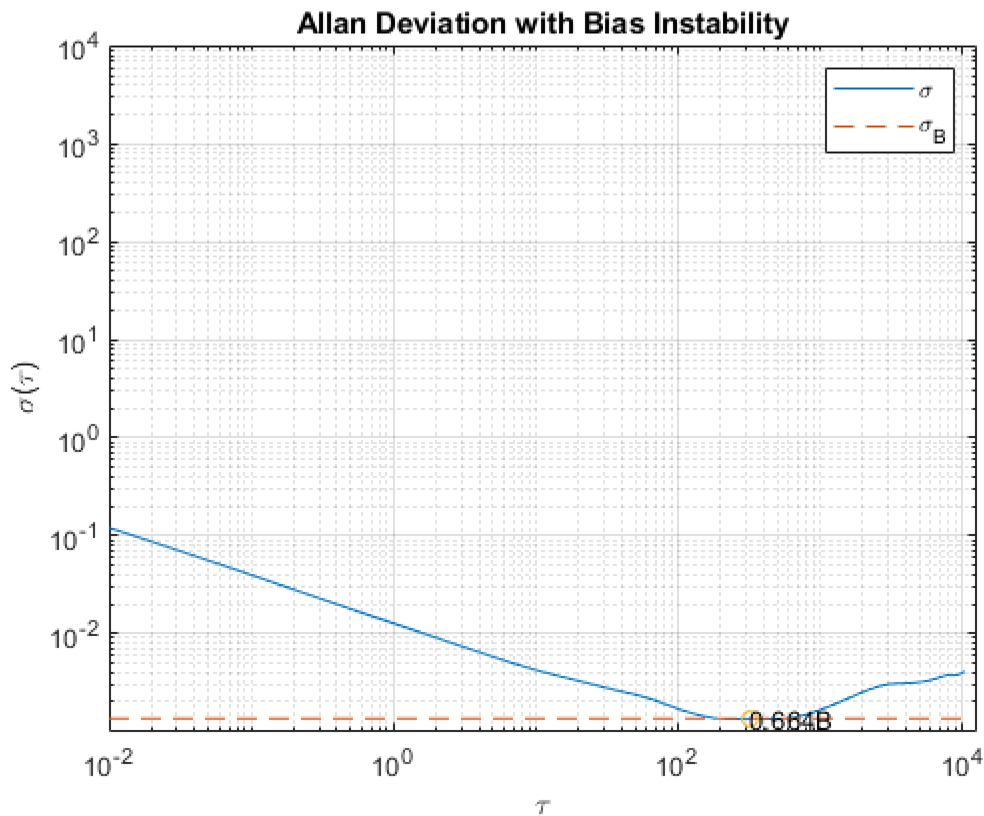
$$\sigma^2(\tau) = \frac{2B^2}{\pi}\ln 2$$

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 $\sqrt{\frac{2\ln 2}{\pi}} \approx 0.664$.

```
% 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
```

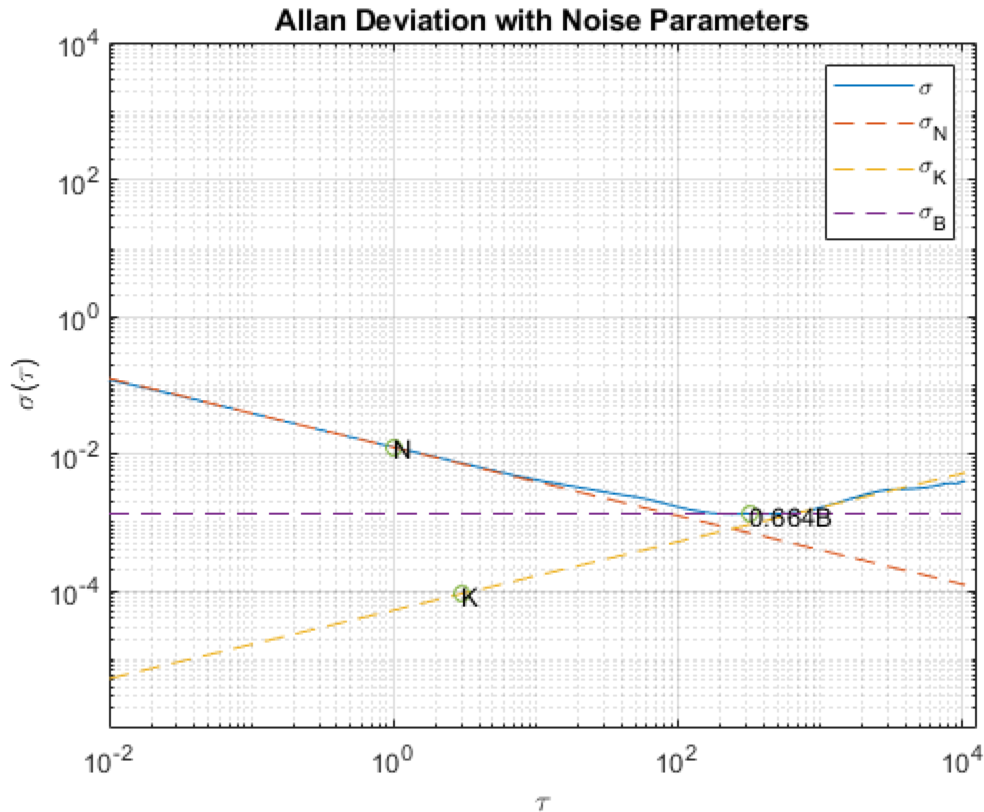
```
B =

    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', '\sigma_N', '\sigma_K', '\sigma_B')
text(tauParams, params, {'N', 'K', '0.664B'})
```

```
grid on
axis equal
```



Gyroscope Simulation

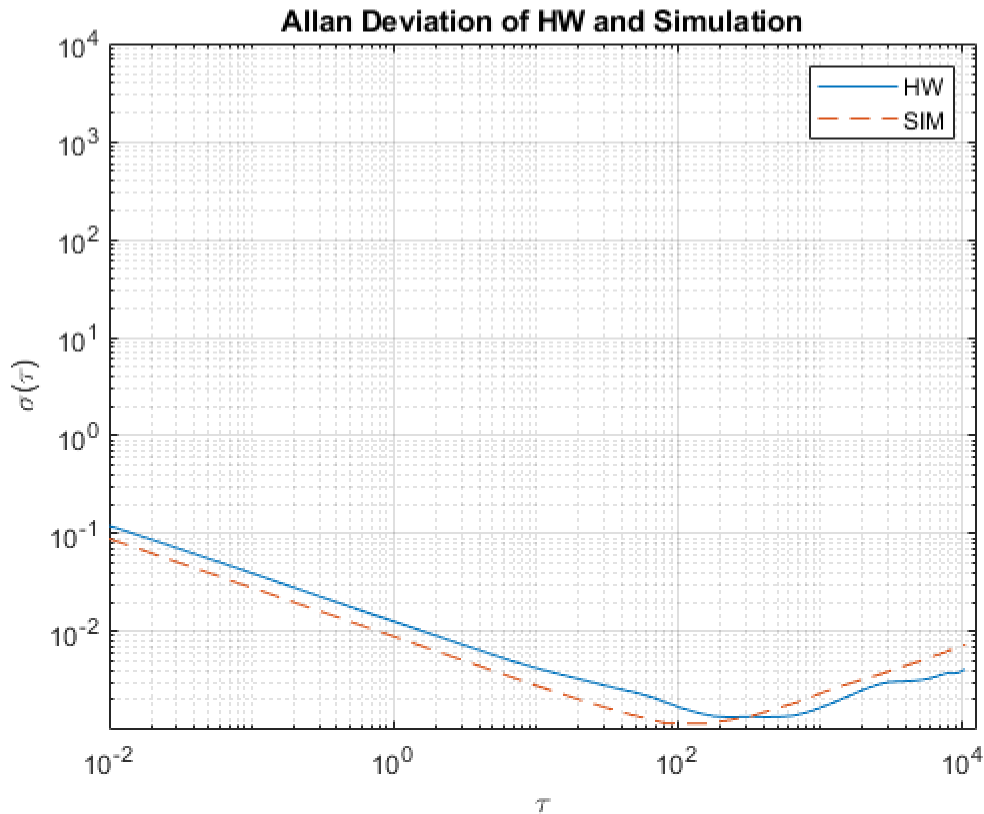
Use the `imuSensor` object to simulate gyroscope measurements with the noise parameters identified above.

```
% 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

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

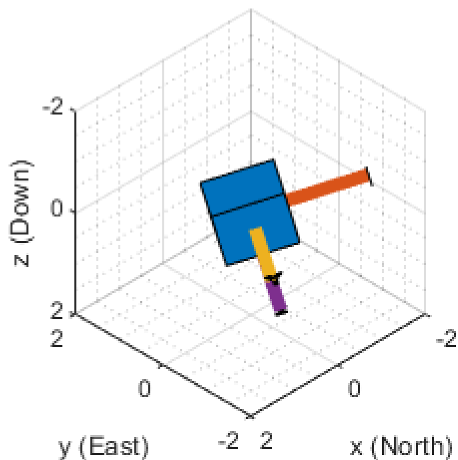
three different axes: first around its local Y-axis, then around its Z-axis, and finally around its X-axis. The device's X-axis was generally pointed southward for the duration of the experiment.

```
ld = load('rpy_9axis.mat');

acc = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;
mag = ld.sensorData.MagneticField;

viewer = HelperOrientationViewer;
```

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)
    viewer(qe(ii));
    pause(0.01);
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 could be used to smooth the motion.

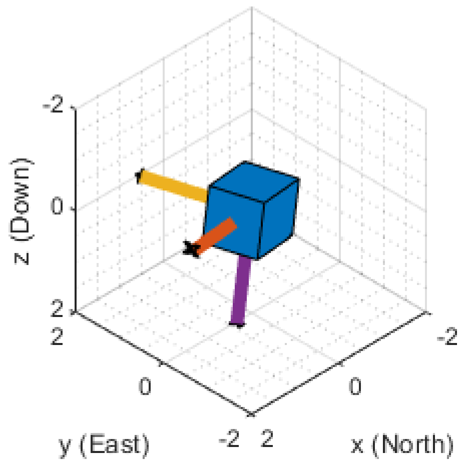
Accelerometer-Gyroscope Fusion

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

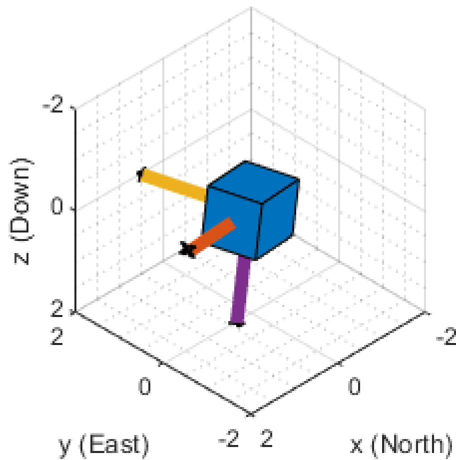
memory constraints, and has minimal tunable parameters, which allows for easier configuration at the cost of finer tuning.

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,:));
    viewer(qimu);
    pause(0.01);
end
```




```
% Disable magnetometer input.
cfilt = complementaryFilter('SampleRate', ld.Fs, 'HasMagnetometer', false);
for ii=1:size(acc,1)
    qimu = cfilt(acc(ii,:), gyro(ii,:));
    viewer(qimu);
    pause(0.01);
end
```



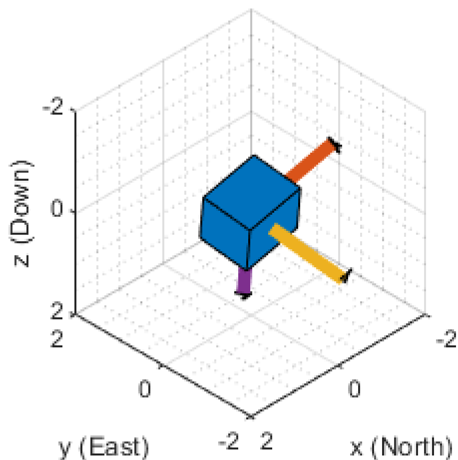
Although the `imufilter` and `complementaryFilter` algorithms produce significantly smoother estimates of the motion, compared to the `ecompass`, they do not correctly estimate the direction of north. The `imufilter` does not process magnetometer data, so it simply assumes the device's X-axis is initially 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

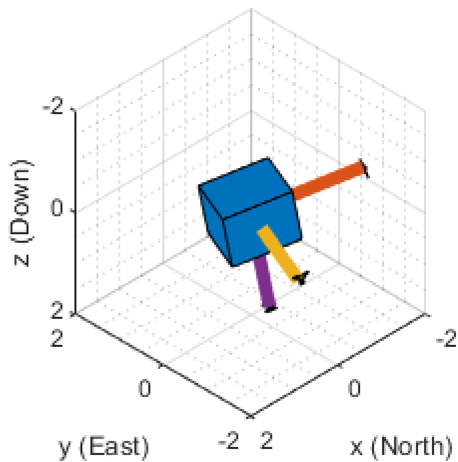
An attitude and heading reference system (AHRS) consists of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation. The `ahrsfilter` and `complementaryFilter` System objects™ combine the best of the previous

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,:));
    viewer(qahrs);
    pause(0.01);
end
```



```
cfilt = complementaryFilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qahrs = cfilt(acc(ii,:), gyro(ii,:), mag(ii,:));
    viewer(qahrs);
    pause(0.01);
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.

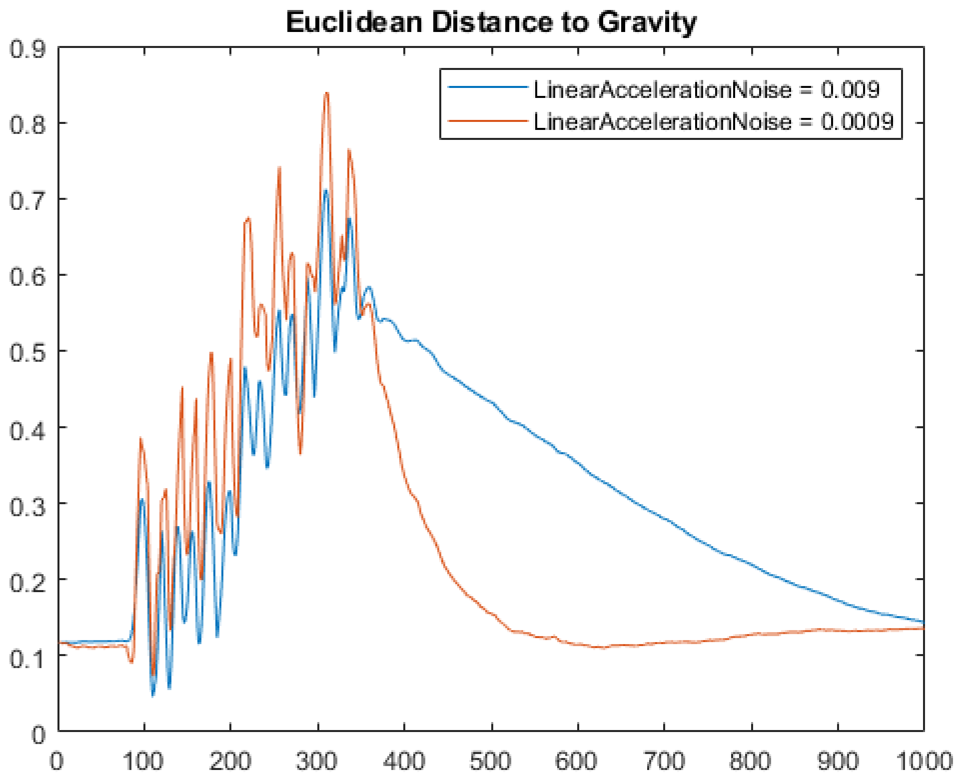
Consider how an imufilter with a LinearAccelerationNoise of 9e-3 $(m/s^2)^2$ responds to a shaking trajectory, compared to one with a LinearAccelerationNoise of 9e-4 $(m/s^2)^2$.

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

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

lowVarFilt = imufilter('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

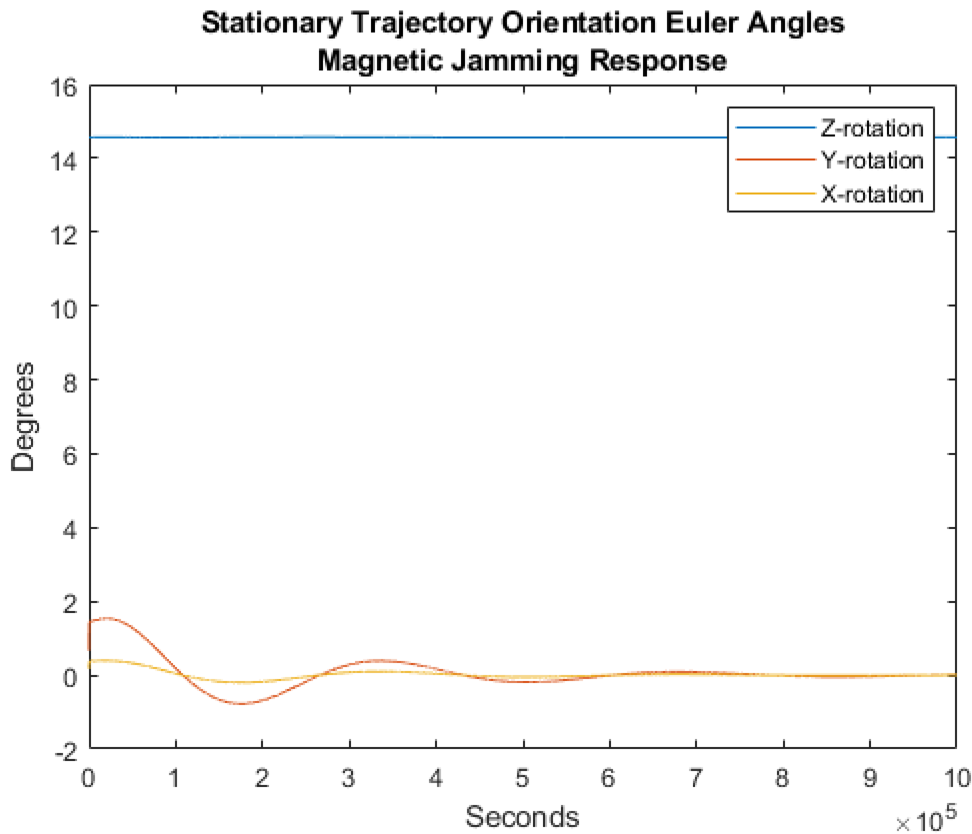
the accelerometer signal. As such, the orientation estimate converges quickly back to vertical once the shaking has ended. The converse is true for `highVarFilt`. The filter is less affected by shaking, but the orientation estimate takes longer to converge to vertical when the shaking has stopped.

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 `insfilter` 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 `insfilter` 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 = insfilter;
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 `insfilter`

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

```
     State                           Units        State Vector Index
 Orientation as a quaternion                      1:4
 Velocity (NED)                      m/s          5:7
 Position (NED)                      m            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 `insfilter`

The `insfilter` 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
    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
        -2, 2
        -2 2
        -2 2]);
end

if usePoseView
    posescope = HelperPoseViewer(...
        'XPositionLimits', [-15 15], ...
        'YPositionLimits', [-15, 15], ...
```

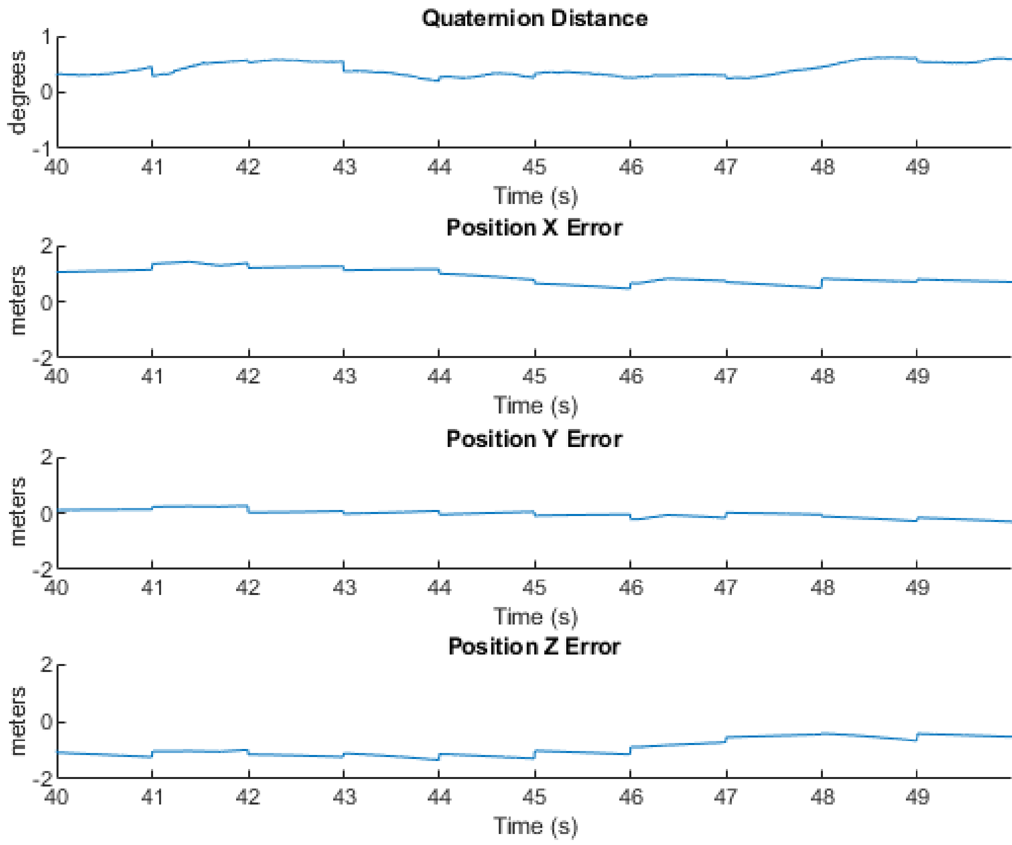
```
        'ZPositionLimits', [-10 10]);
end
```

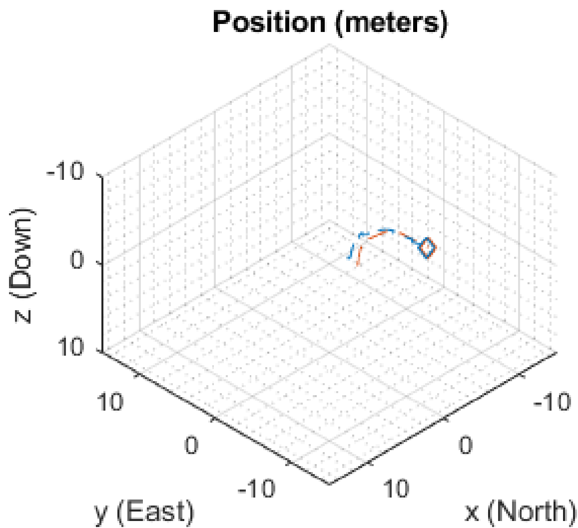
Simulation Loop

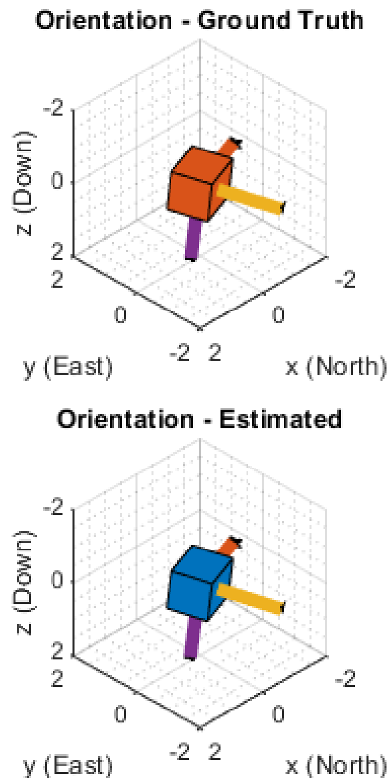
The main simulation loop is a while loop with a nested for loop. The while loop executes at `gpsFs`, which is the GPS sample rate. The nested for loop executes at `imuFs`, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

```
% 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,:);
```

```
                errscope(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
```


Quaternion Distance
degrees
1
0
-1
40
41
42
43
44
45
46
47
48
49
Time (s)
Position X Error
meters
2
0
-2
Time (s)
Position Y Error
meters
Time (s)
Position Z Error
meters
Time (s)





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.

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

fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));
```

```
End-to-End Simulation Position RMS Error
    X: 0.57 , Y: 0.53, Z: 0.68   (meters)

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

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.

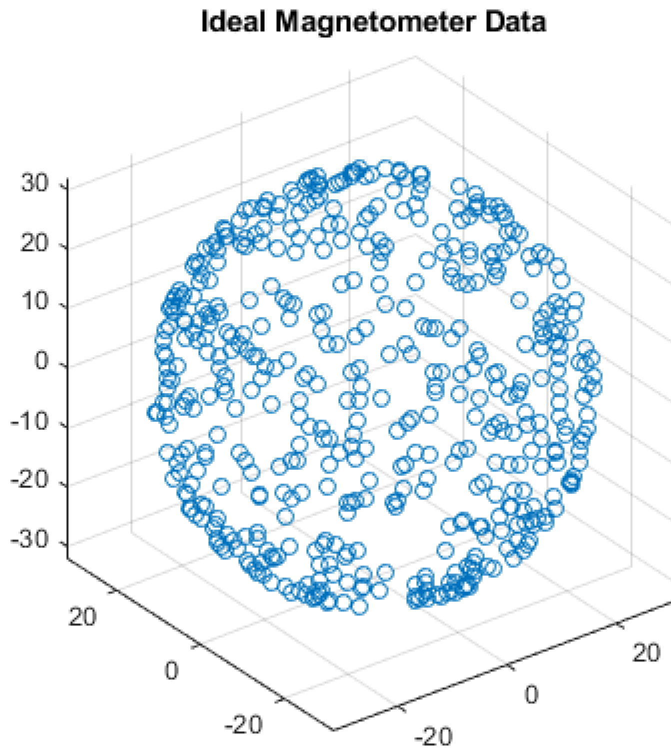
In order to be useful for heading and orientation computation, typical low cost MEMS magnetometers need to be calibrated to compensate for environmental noise and manufacturing defects.

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.

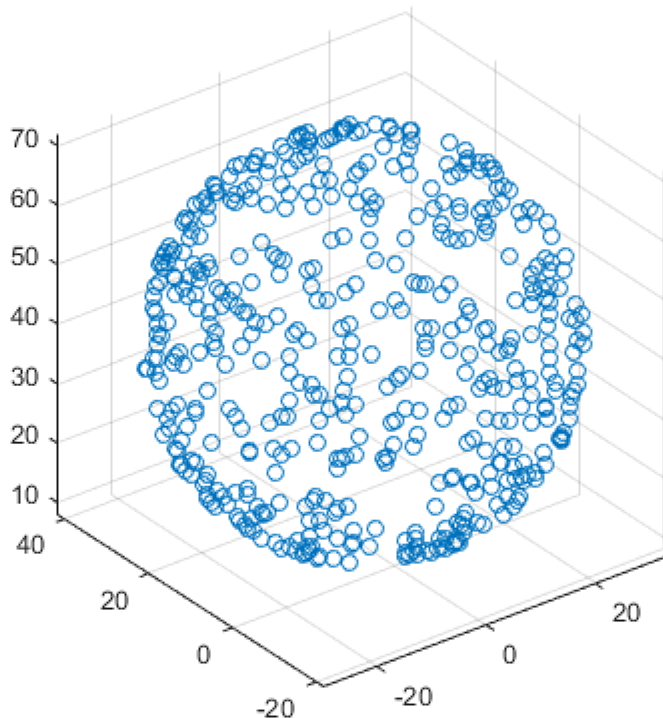
```
N = 500;
rng(1);
acc = zeros(N,3);
av = zeros(N,3);
q = randrot(N,1); % uniformly distributed random rotations
imu = imuSensor('accel-mag');
[~,x] = imu(acc,av,q);
scatter3(x(:,1),x(:,2),x(:,3));
axis equal
title('Ideal Magnetometer Data');
```



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');
```



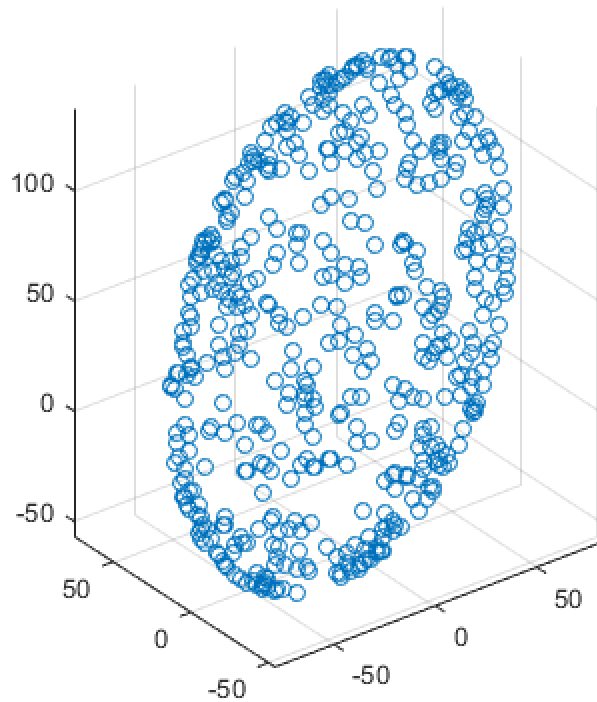
Soft Iron Effects

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');
```



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.

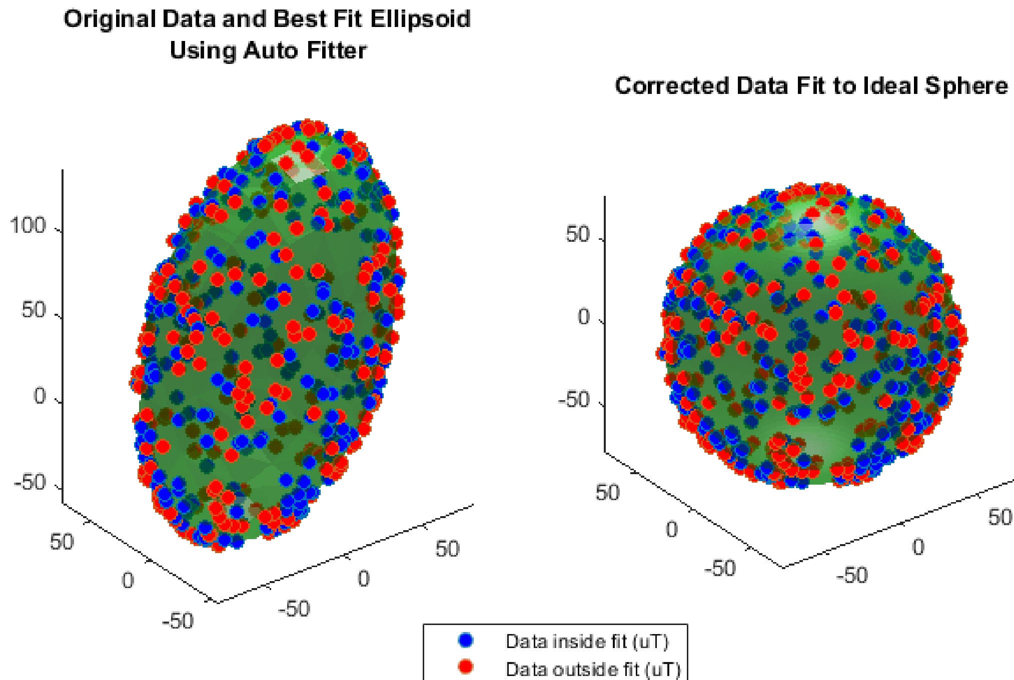
```
imu.Magnetometer.NoiseDensity = 0.08;
for ii=1:numel(q)
    imu.MagneticField = soft(ii,:);
    [~,x(ii,:)] = imu(acc(ii,:),av(ii,:),q(ii));
end
```

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);
```

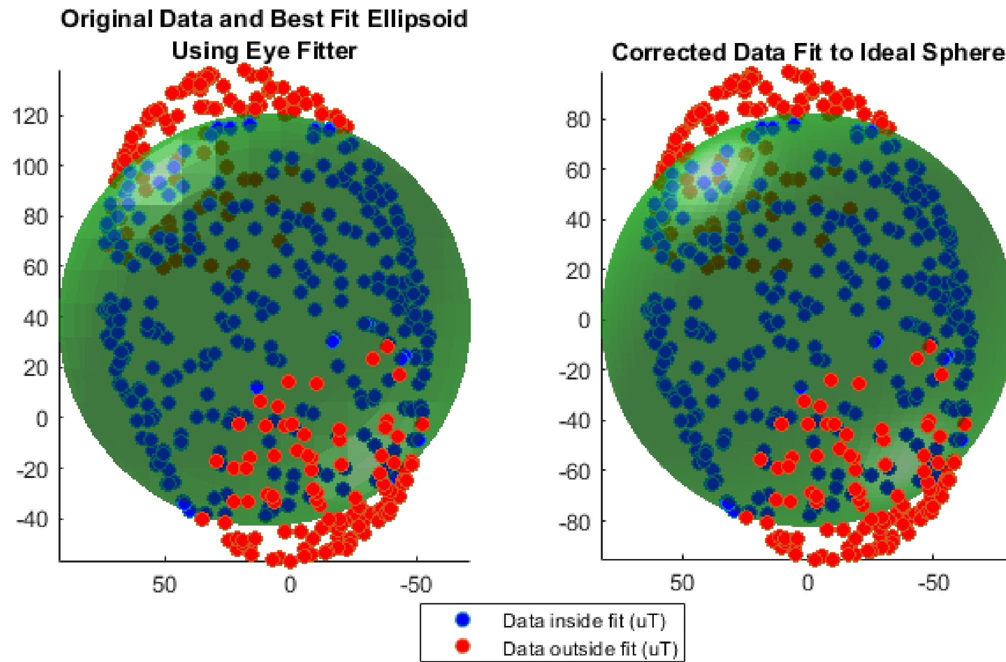
```
Residual error in corrected data : 0.01
```

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:

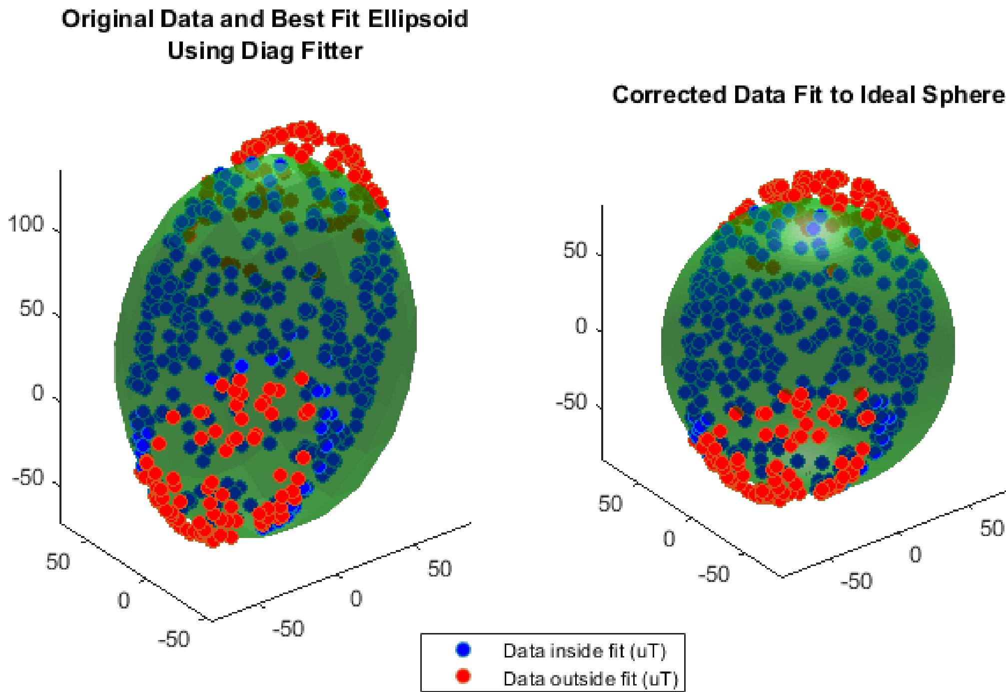
```
[Aeye,beye,expMFSeye] = magcal(x,'eye');
xEyeCorrected = (x-beye)*Aeye;
[ax1,ax2] = de.plotCalibrated(Aeye,beye,expMFSeye,x,xEyeCorrected,'Eye');
view(ax1,[-1 0 0]);
view(ax2,[-1 0 0]);
```



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 semiaxes 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 + 0.6221i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.7334i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   2.1919 + 0.0000i
```

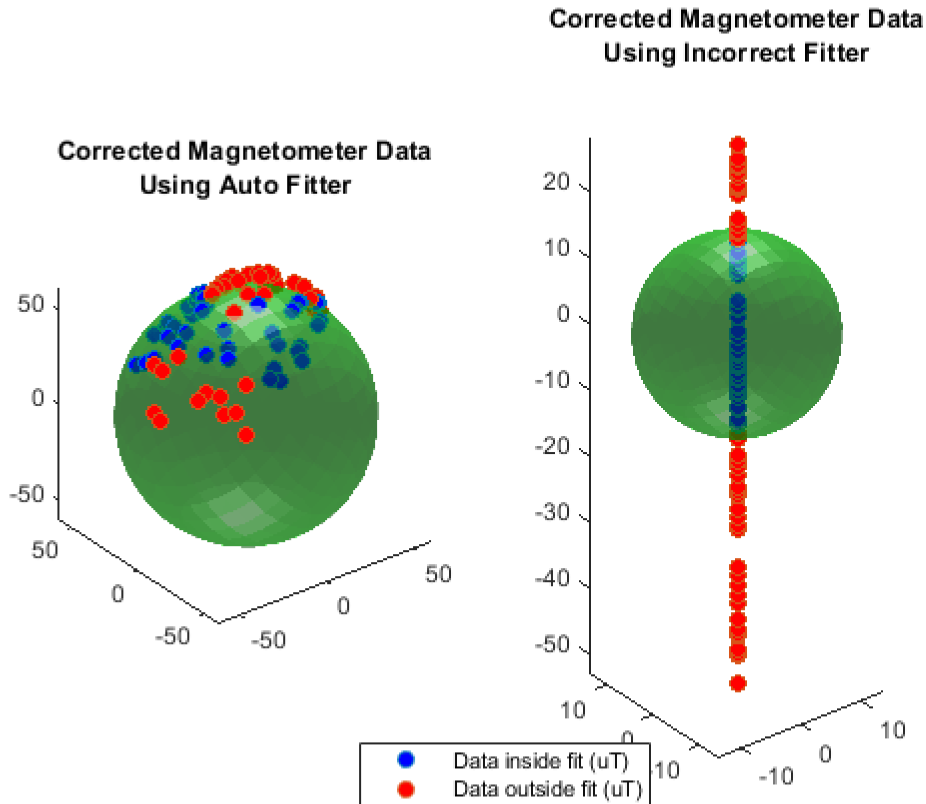
Using the `'auto'` fit option avoids this problem and finds a simpler `A` matrix which is real, symmetric, and positive definite. Calling `magcal` with the `'auto'` option string is the same as calling without any option string.

```
[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,mfspoor,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.

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);
```

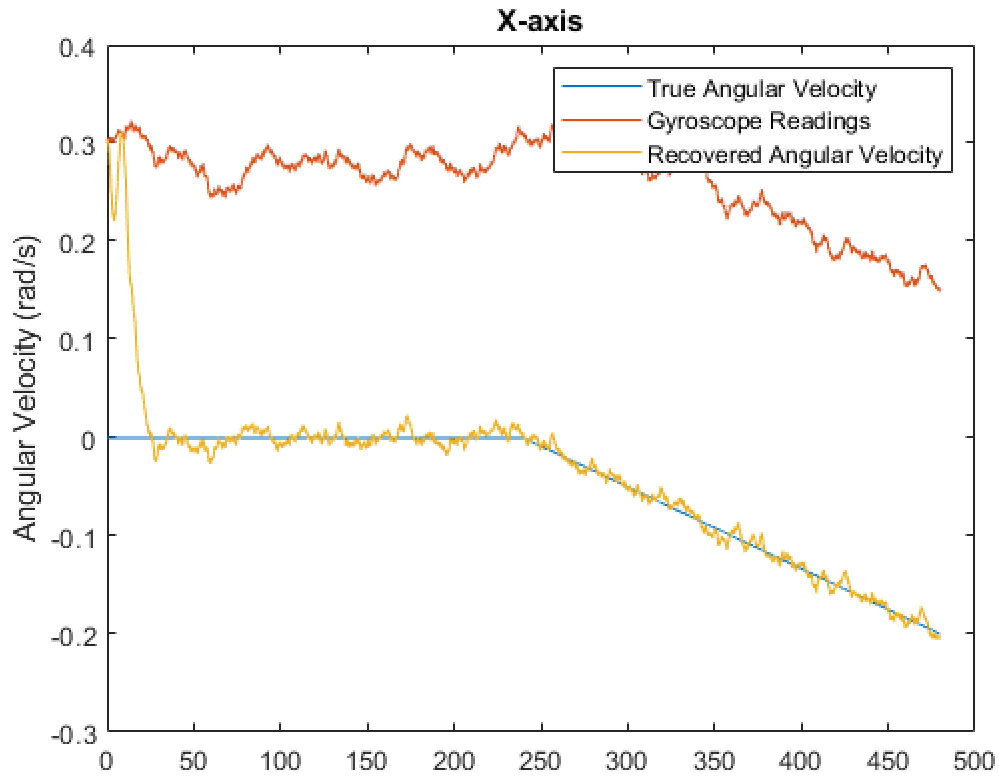
Create an `imufilter` System object, `fuse`. Call `fuse` with the modeled accelerometer readings and gyroscope readings.

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

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

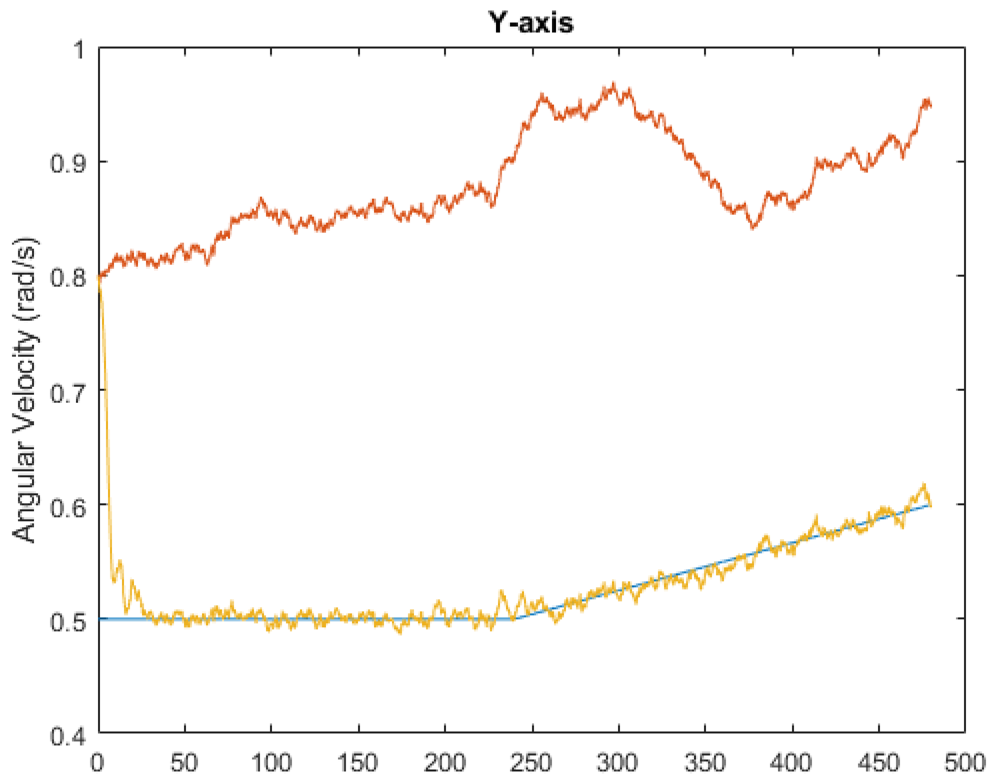
Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

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

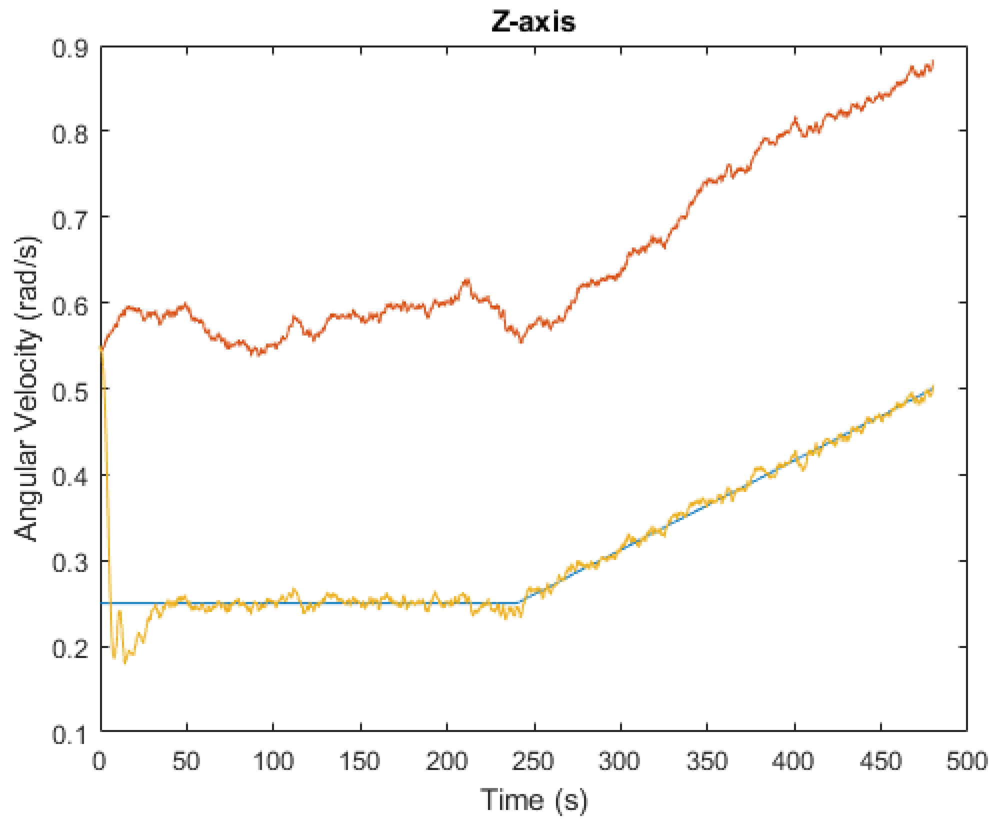
```
time = (0:numSamples-1)'/fs;

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)')
```


Z-axis
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Angular Velocity (rad/s)
0
50
100
150
200
250
300
350
400
450
500
Time (s)

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

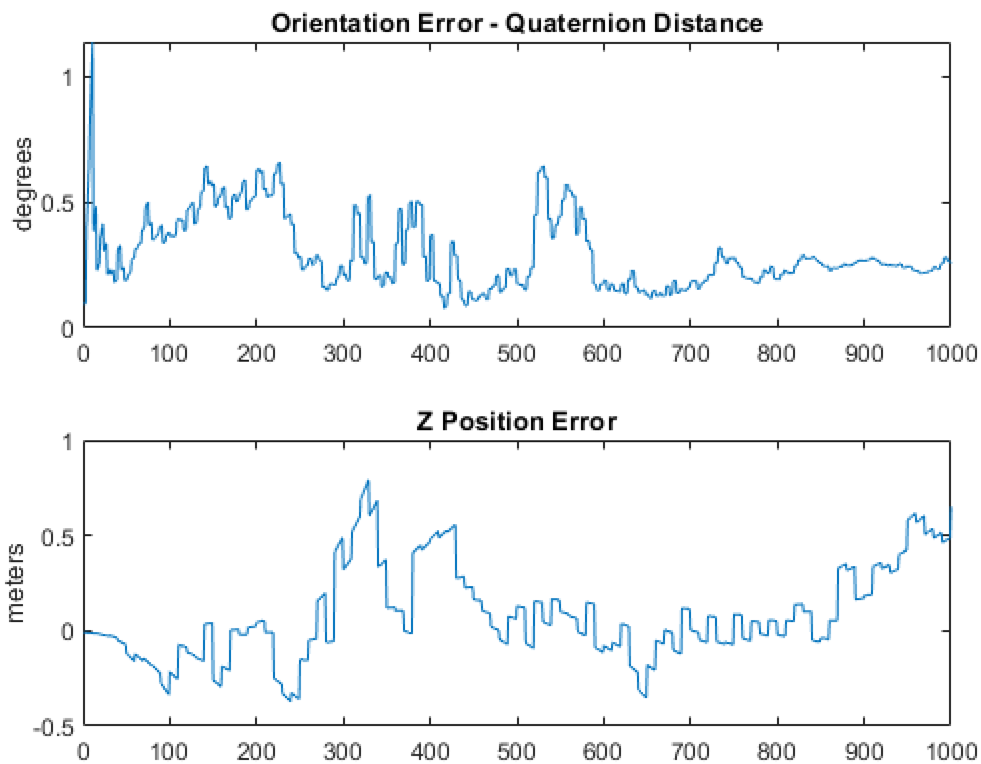
```
q = initOrient;
firstTime = true;

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 the `ahrs10filter` to perform 10-axis sensor fusion for height and orientation.

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.

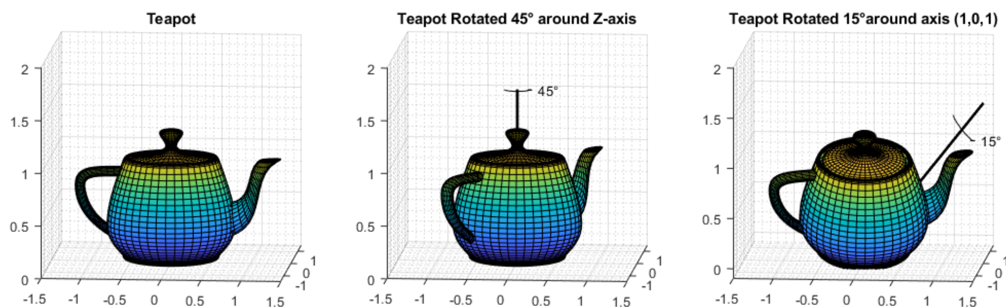
The `HelperDrawRotation` class is used to illustrate several portions of this example.

```
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.

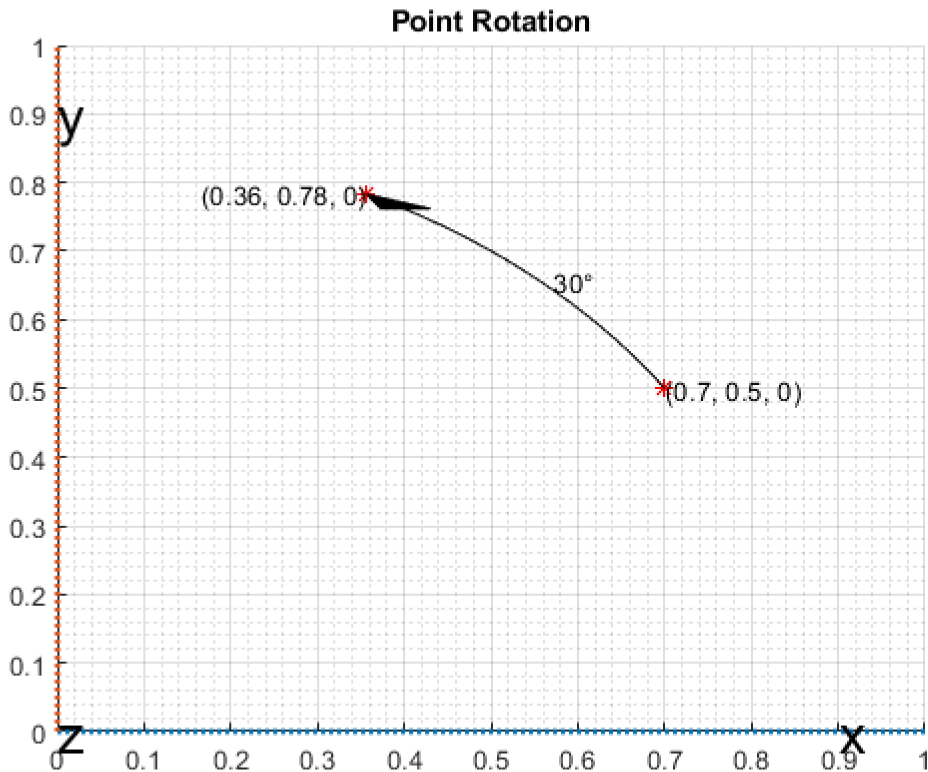
```
dr.drawTeapotRotations;
```



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.

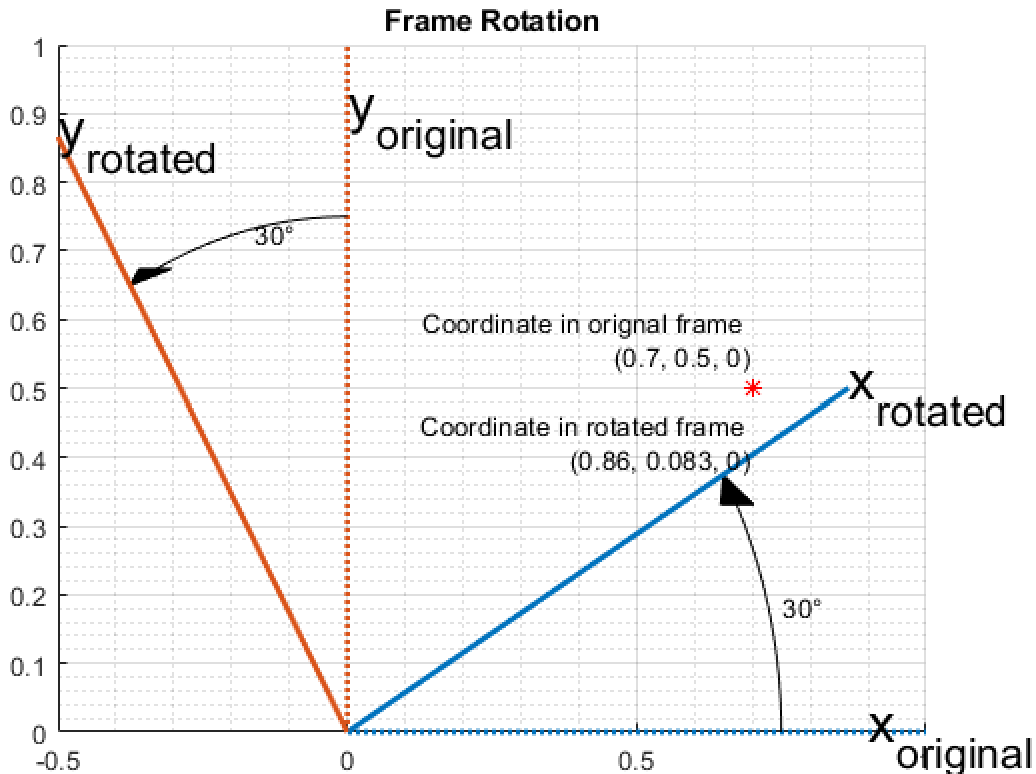
```
figure;
dr.draw2DPointRotation(gca);
```



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);
```

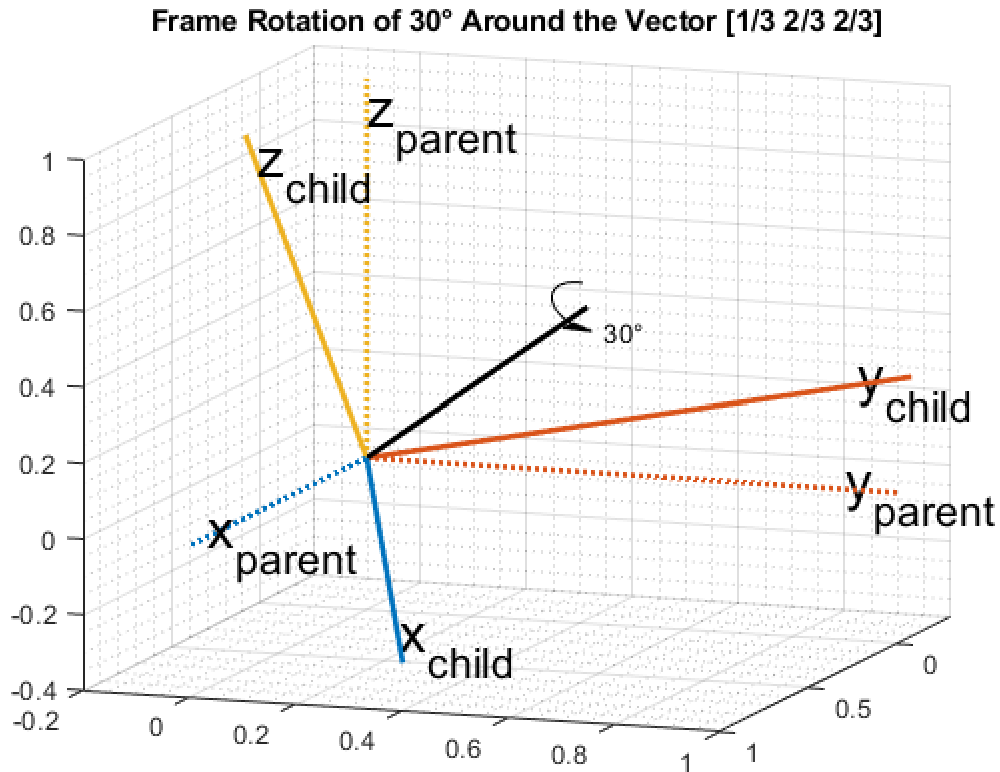


Orientation

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

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

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

$$\cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(x\mathbf{i} + y\mathbf{j} + z\mathbf{k})$$

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

$$norm(q) = \sqrt{a^2 + b^2 + c^2 + d^2}$$

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.\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)
```

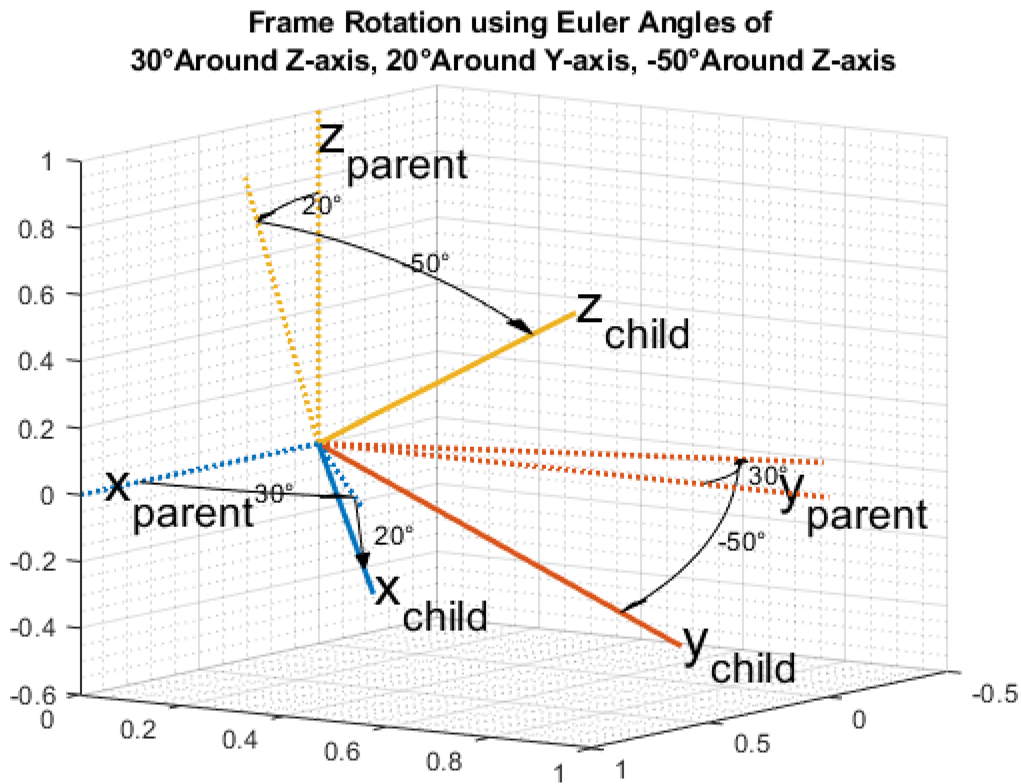
```
ans =

    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 `pi`. 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:

cat                 ismatrix            power
classUnderlying     isnan               prod
compact             isrow               quaternion
conj                isscalar            rdivide
ctranspose          isvector            reshape
disp                ldivide             rotateframe
dist                length              rotatepoint
double              log                 rotmat
eq                  meanrot             rotvec
euler               minus               rotvecd
eulerd              mtimes              single
exp                 ndims               size
horzcat             ne                  slerp
iscolumn            norm                times
isempty             normalize           transpose
isequal             numel               uminus
isequaln            parts               validateattributes
isfinite            permute             vertcat
isinf               plus
```

```
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:

$$Slerp(q_0, q_1, h) = q_1(q_1^{-1}q_2)^h$$

An alternate formulation with sinusoids (used in the `slerp` function implementation) is:

$$Slerp(q_0, q_1, h) = \frac{\sin((1-h)\theta)}{\sin\theta}q_0 + \frac{\sin(h\theta)}{\sin\theta}q_1$$

where θ is the dot product of the quaternion parts. Note that $\theta = dist(q_0, q_1)/2$.

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 `q0` to `q1`, 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 `q0` and `q1`, 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 `q0` and `q1`, 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.

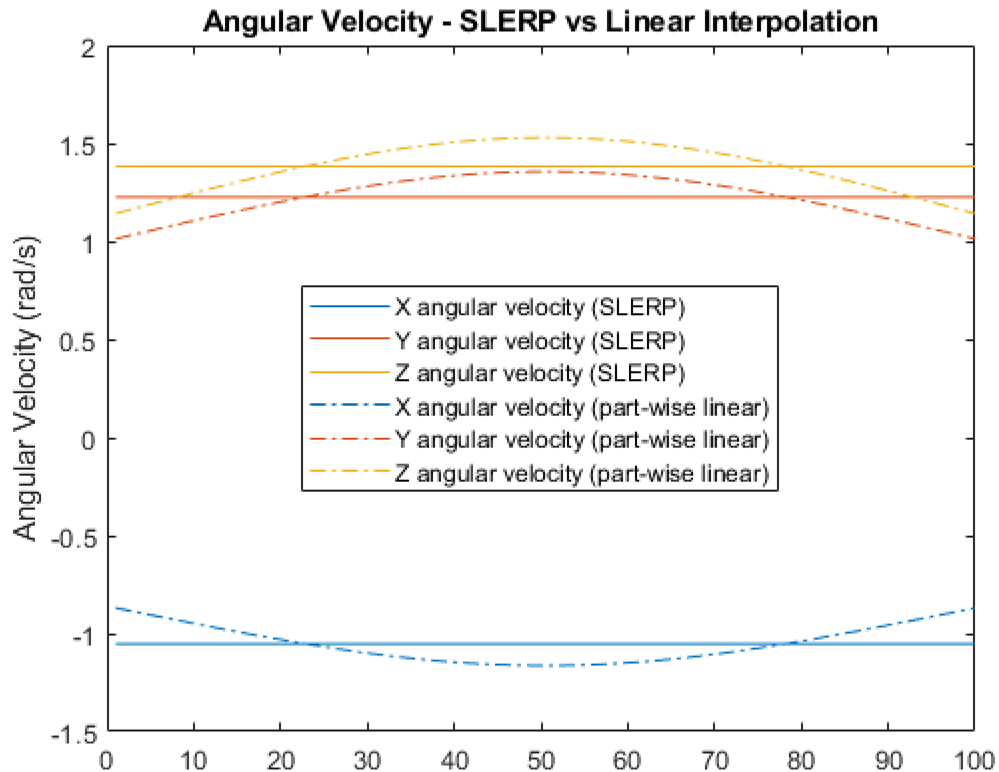
```
trajLerped = normalize(quaternion(partsLinInterp));
```

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;
```

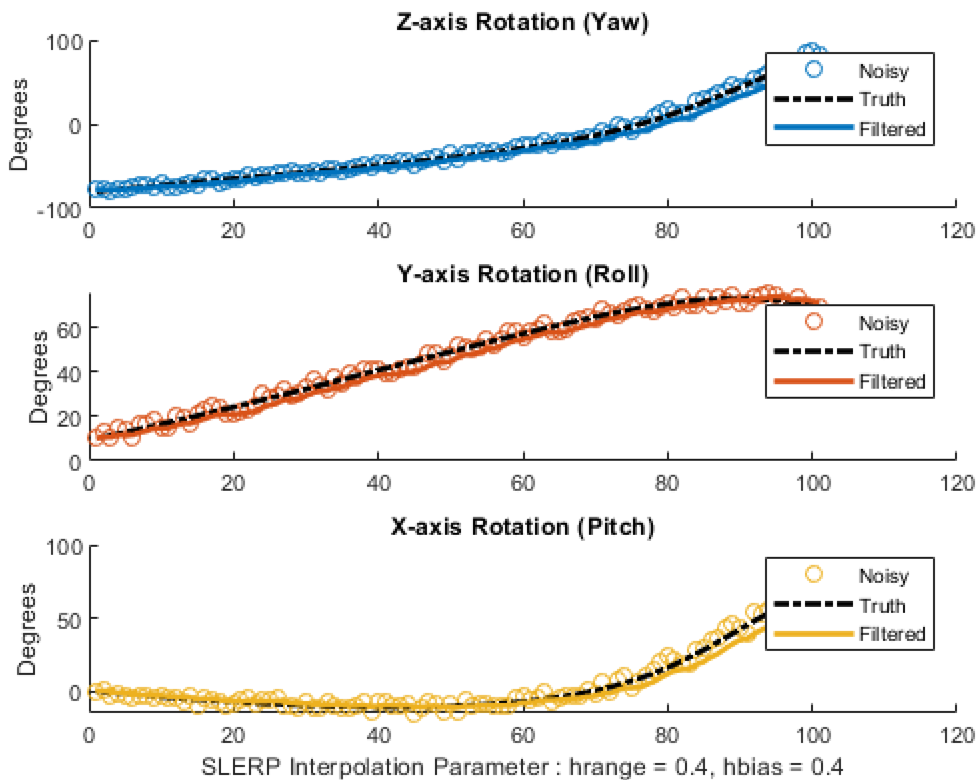
Initialize the filter and preallocate outputs.

```
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 SIGRAPH 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` 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`, `accelparams`, and `magparams`, respectively.

In the following plots, unless otherwise noted, only the x-axis measurements are shown.

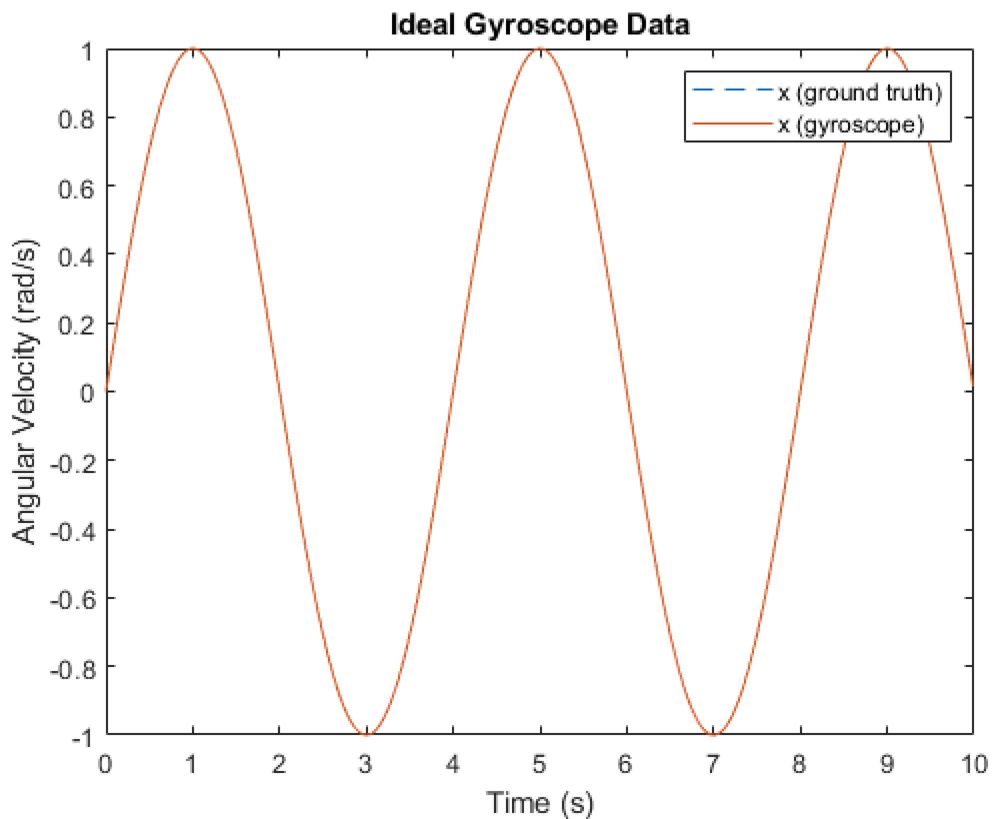
Default Parameters

The default parameters for the gyroscope model simulate an ideal signal. Given a sinusoidal input, the gyroscope output should match exactly.

```
params = gyroparams

% 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)')
```

```
params = 

  gyroparams with properties:

    MeasurementRange: Inf        rad/s
```

```
         Resolution: 0          (rad/s)/LSB
       ConstantBias: [0 0 0]    rad/s
   AxesMisalignment: [0 0 0]    %

      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)
```

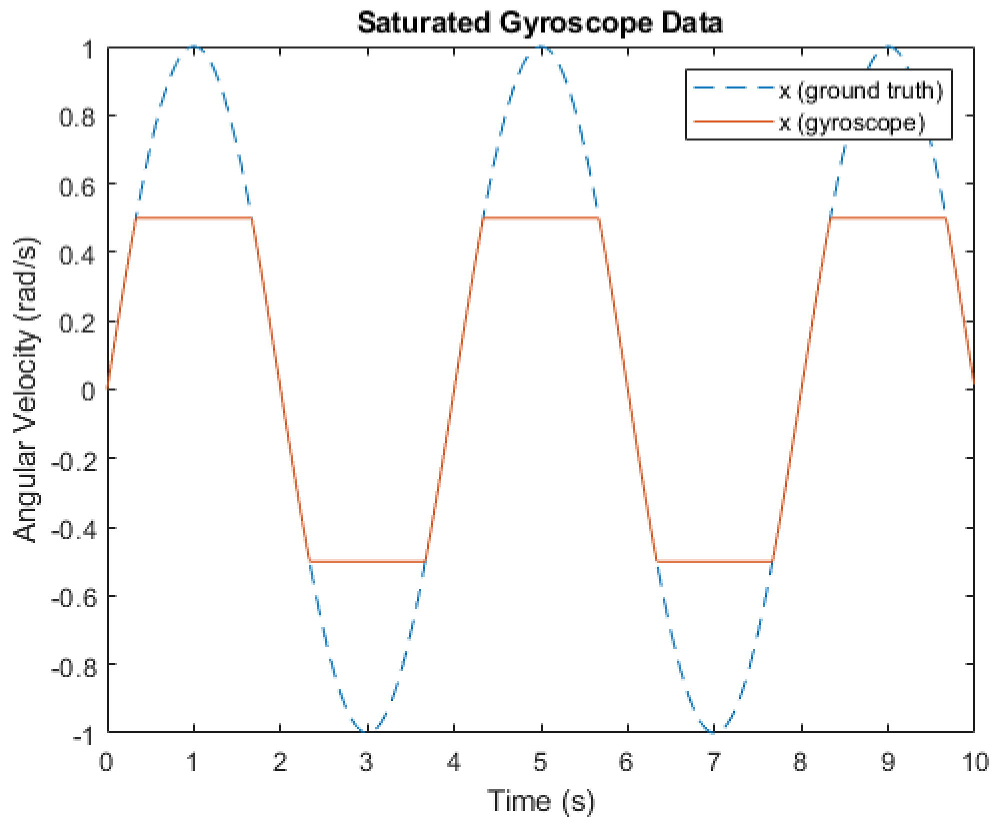


Hardware Parameter Tuning

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.

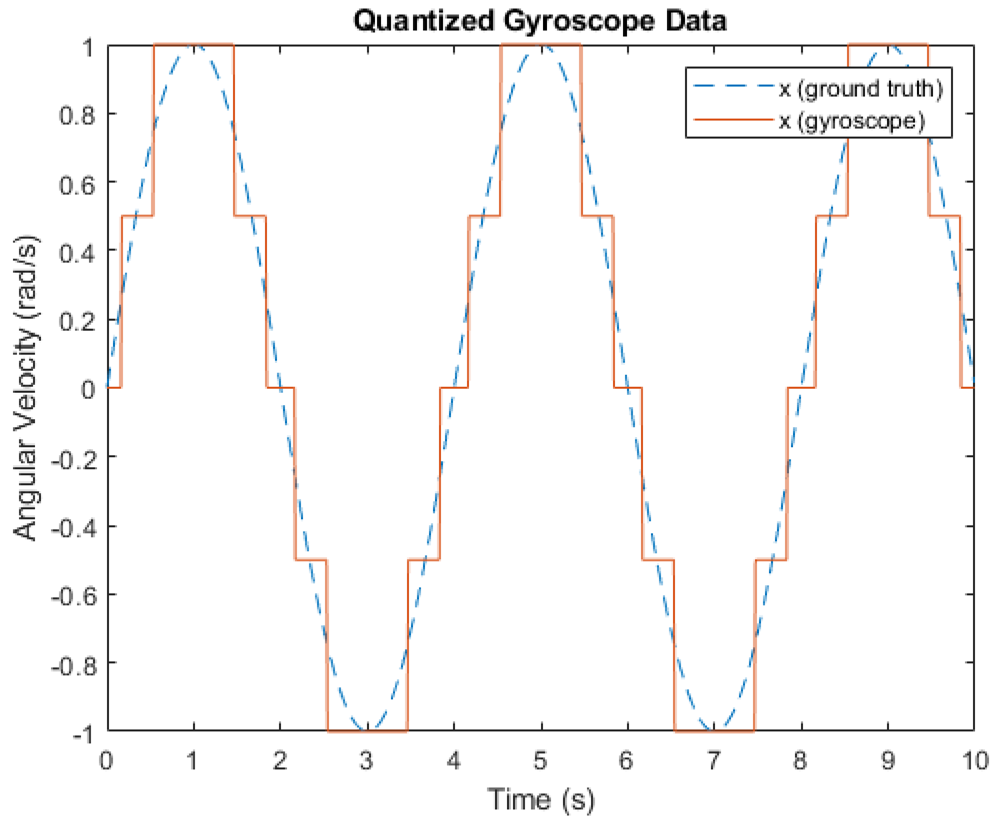
```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.MeasurementRange = 0.5; % rad/s

[~, 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)')
```

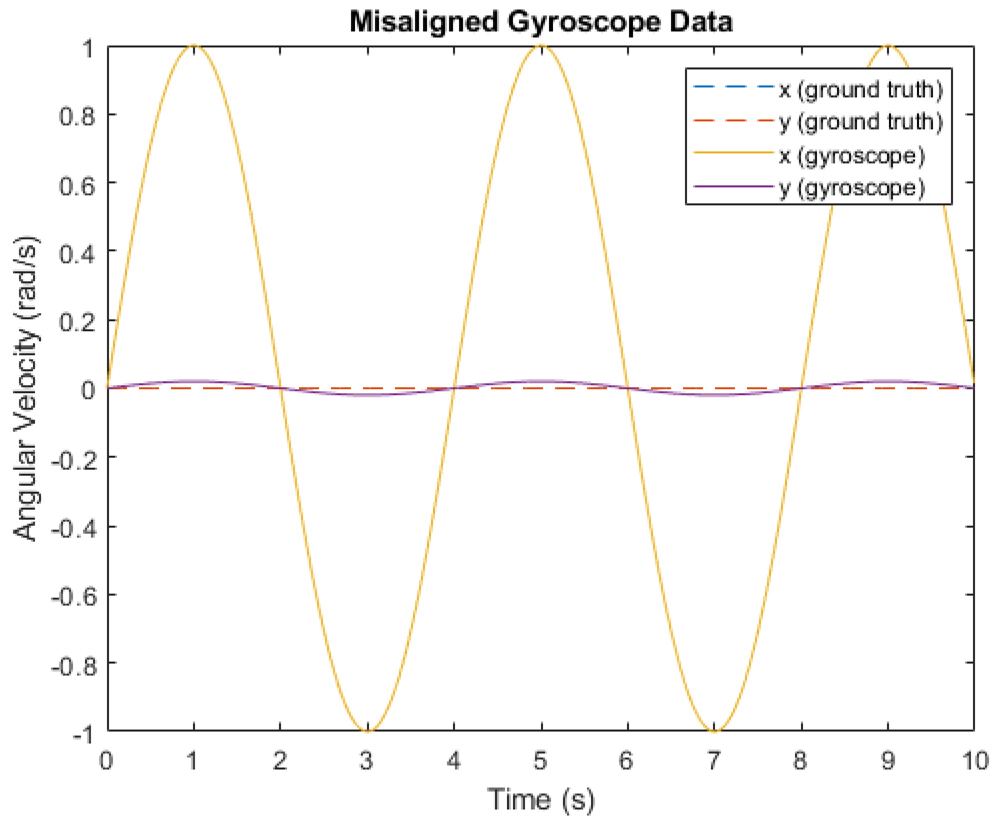
```
title('Quantized Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```



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);
imu.Gyroscope.AxesMisalignment = [2 0 0]; % percent

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1:2), '--', t, gyroData(:,1:2))
```

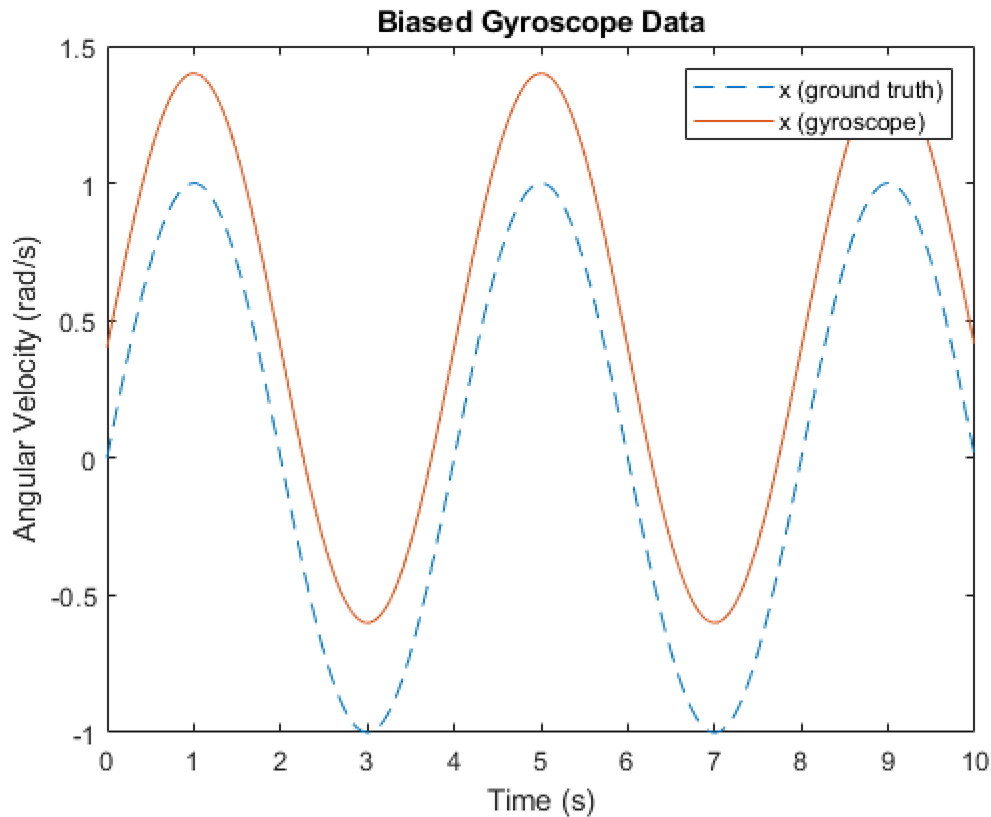
```
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Misaligned Gyroscope Data')
legend('x (ground truth)', 'y (ground truth)', ...
    'x (gyroscope)', 'y (gyroscope)')
```



`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);
imu.Gyroscope.ConstantBias = [0.4 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)')
```

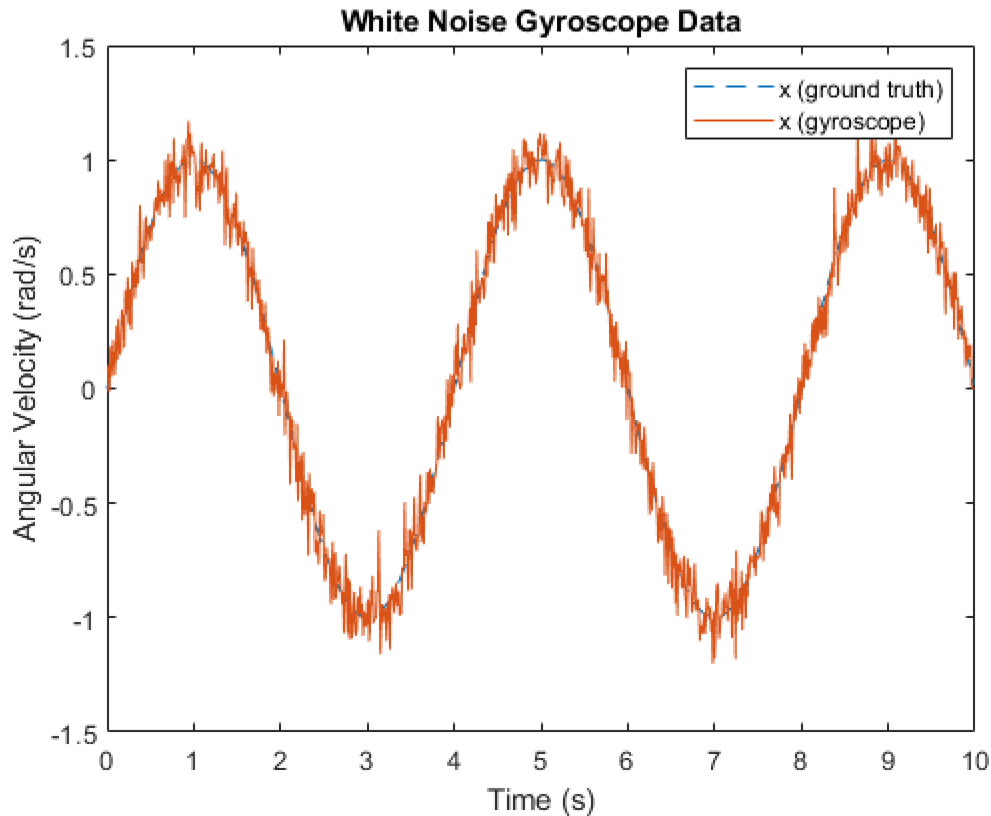


Random Noise Parameter Tuning

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 example.

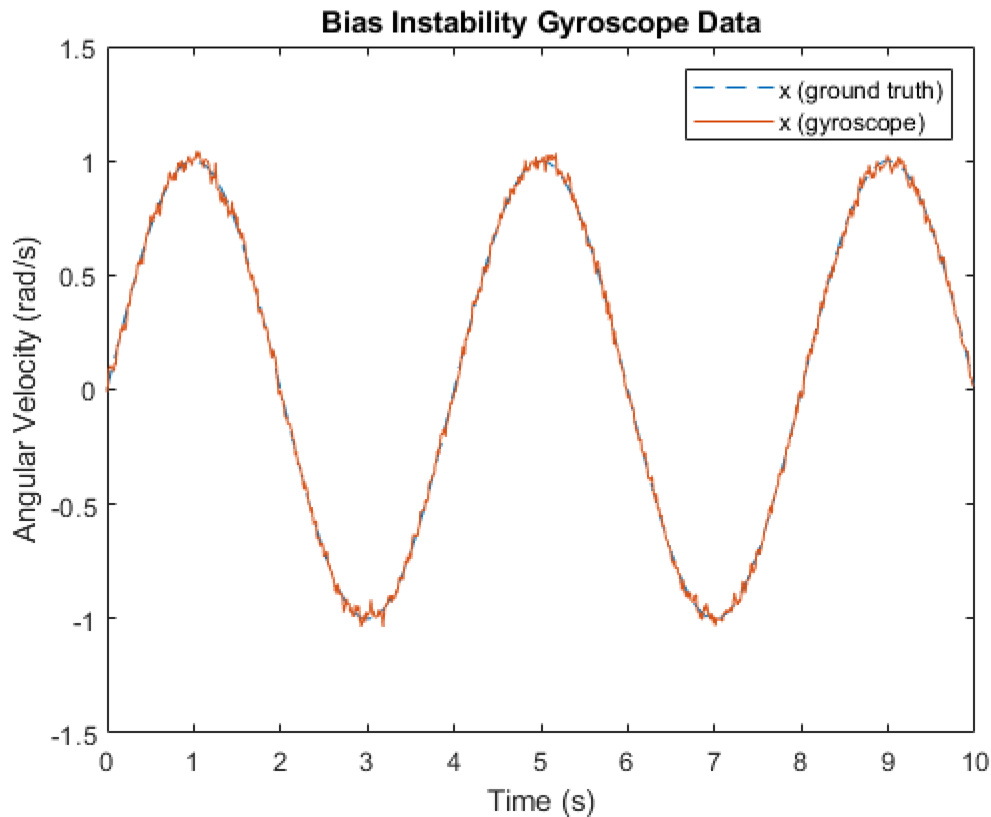
`NoiseDensity` is the amount of white noise in the sensor measurement. It is sometimes called angle random walk for gyroscopes or velocity random walk for accelerometers.

```
rng('default')

imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.NoiseDensity = 1.25e-2; % (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)')
```



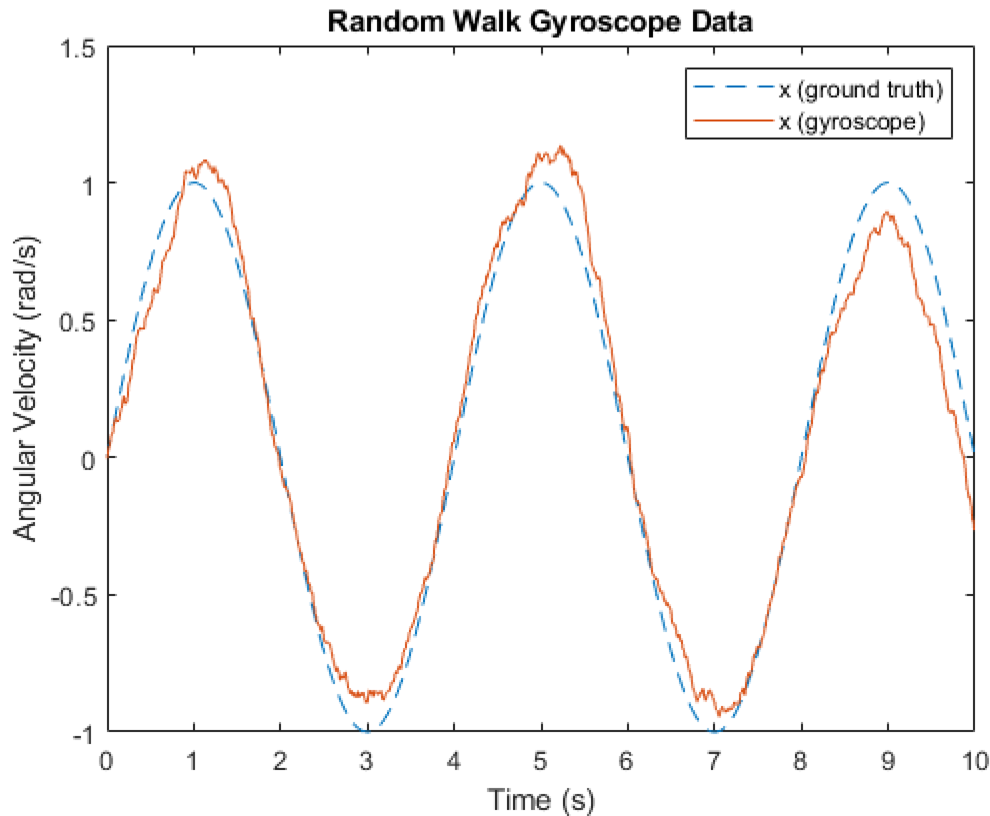
`BiasInstability` is the amount of pink or flicker noise in the sensor measurement.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.BiasInstability = 2.0e-2; % 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 = 9.1e-2; % (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)')
```



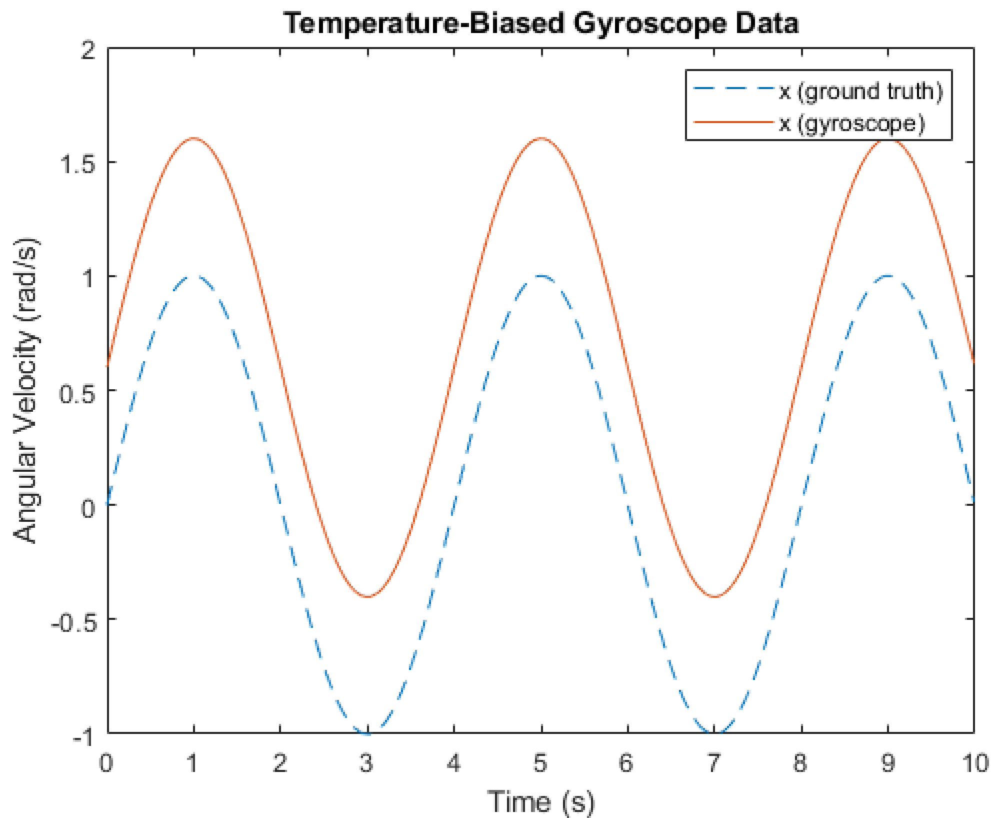
Environmental Parameter Tuning

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.

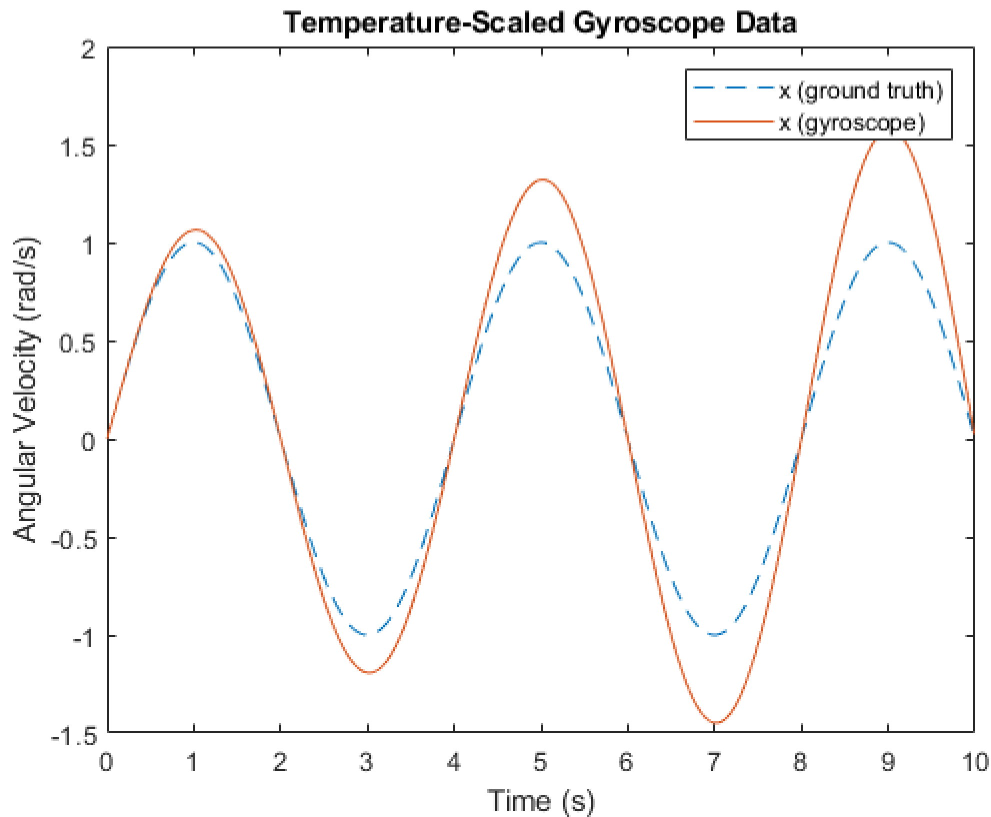
```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureBias = 0.06; % (rad/s)/(degrees C)
imu.Temperature = 35;
```

```
[~, 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.

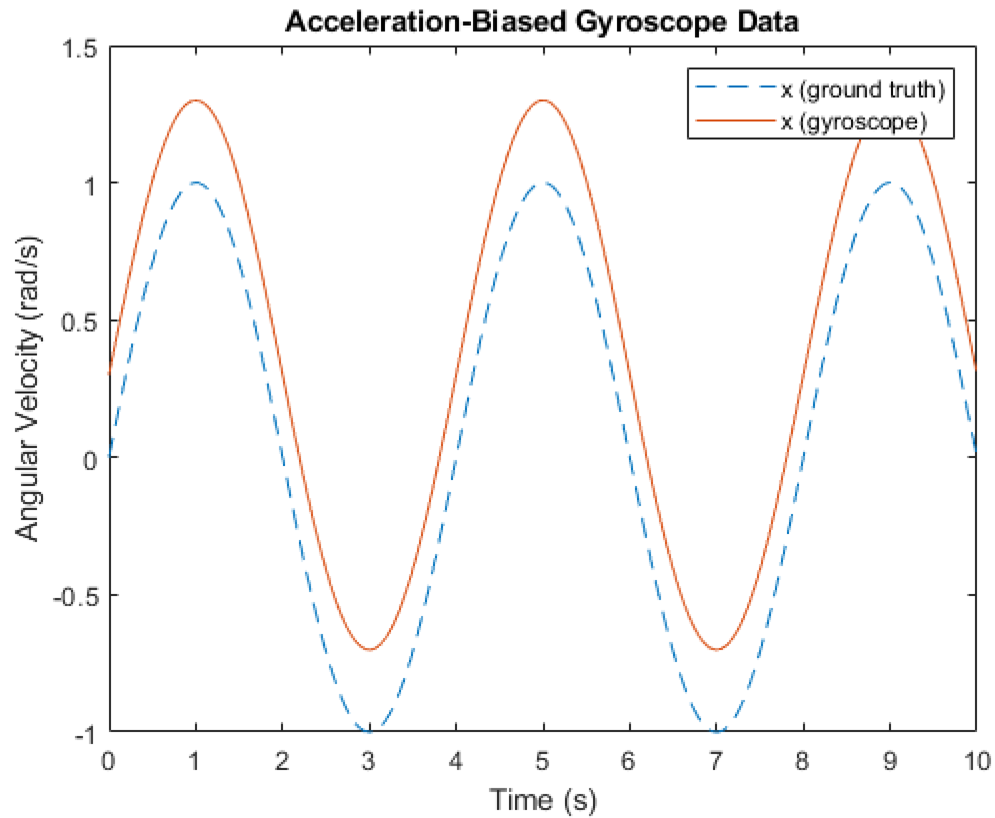
```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureScaleFactor = 3.2; % %/(degrees C)

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

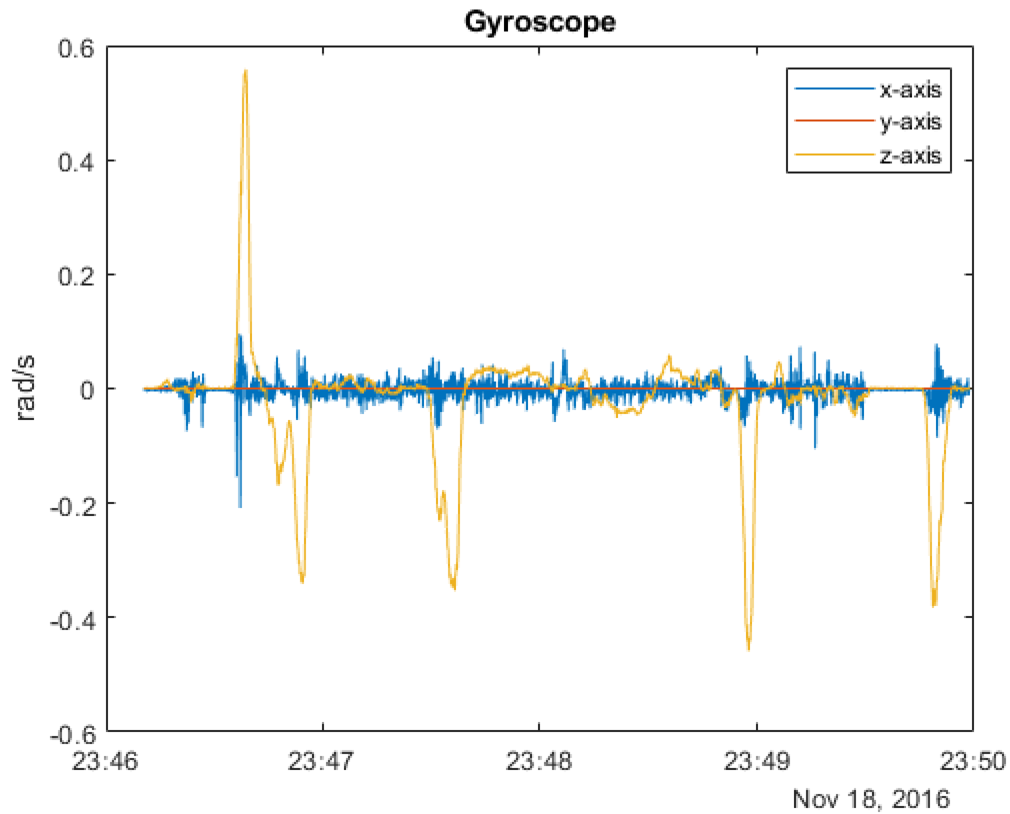
Load logged inertial measurement unit (IMU) data and extract individual sensor data and timestamps.

```
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.

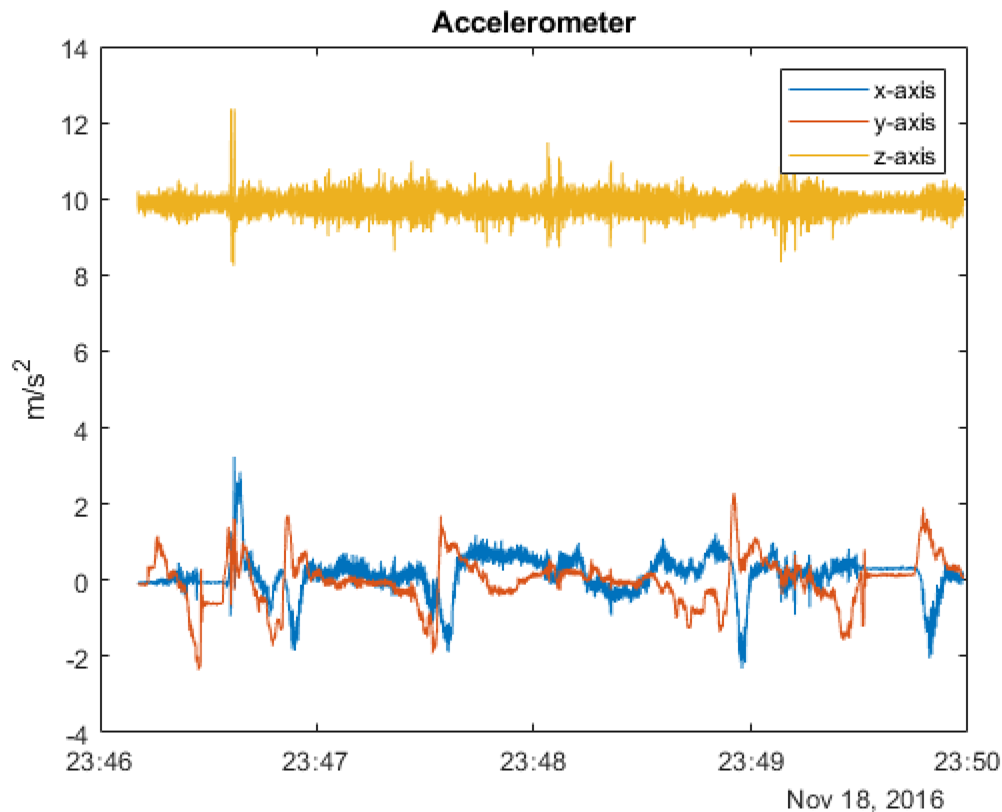
```
figure
plot(time, gyro)
title('Gyroscope')
ylabel('rad/s')
legend('x-axis', 'y-axis', 'z-axis')
```



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')
```

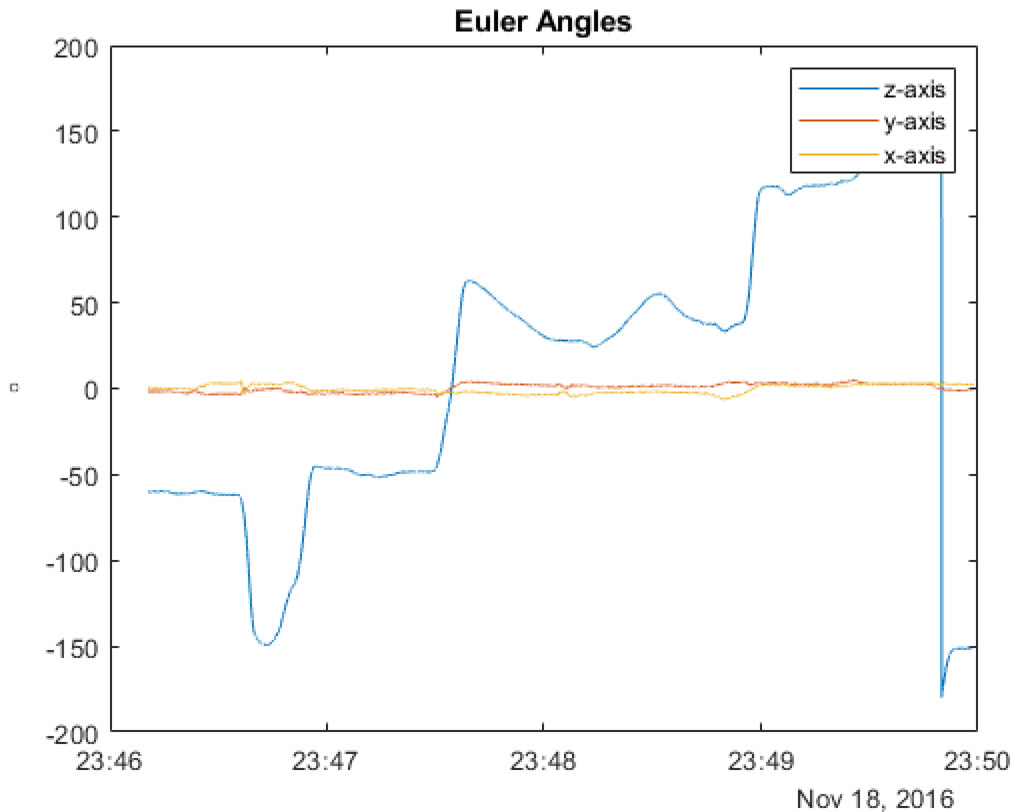


Inspect the Orientation Data

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. Therefoe, 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')
```

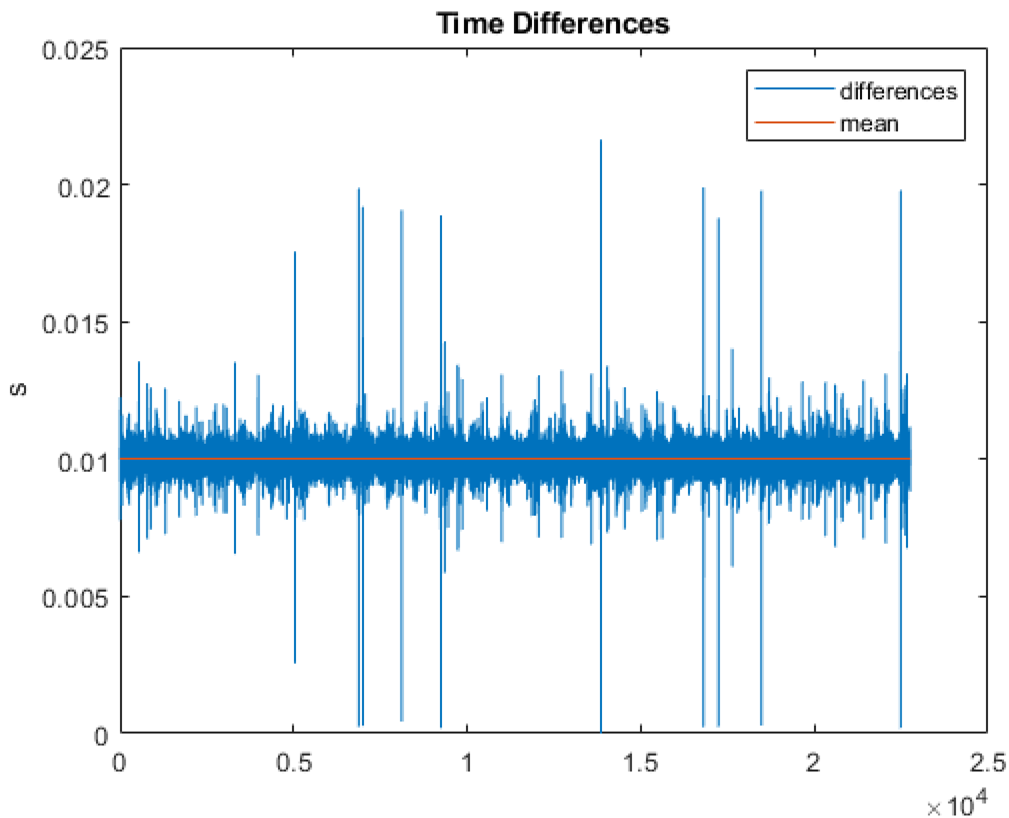


Find the Sampling Rate of the Logged Data

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 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);
```

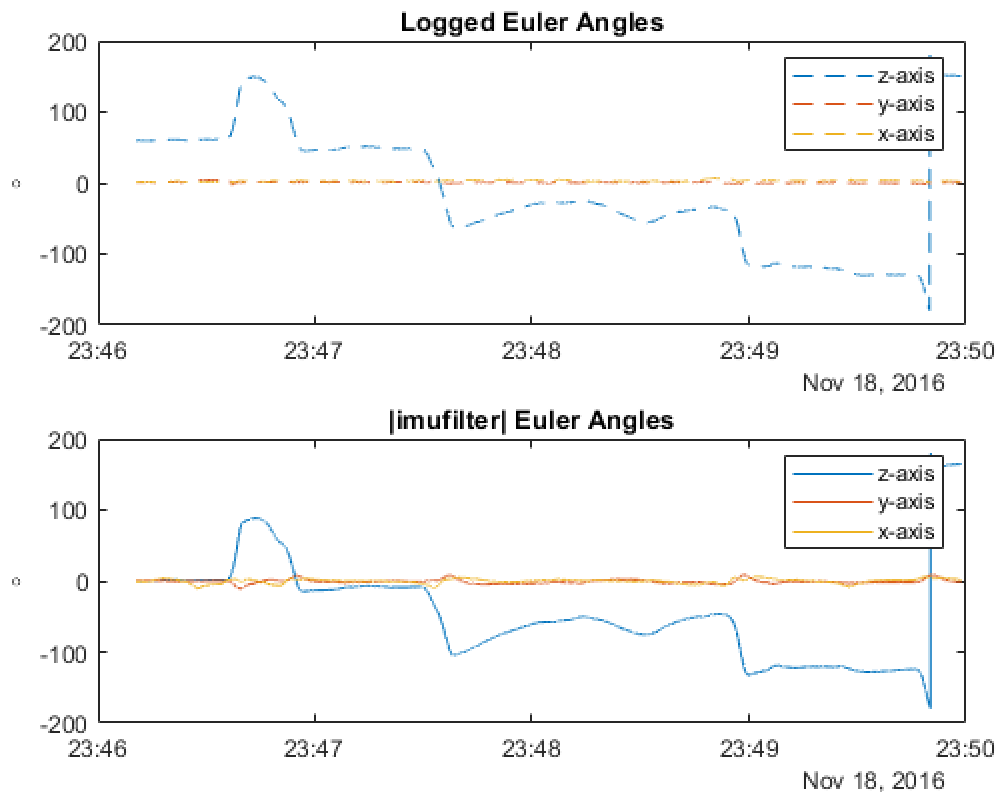
```
figure
plot([deltaTimes, repmat(mean(deltaTimes), numel(deltaTimes), 1)])
title('Time Differences')
ylabel('s')
legend('differences', 'mean')
```



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')
```

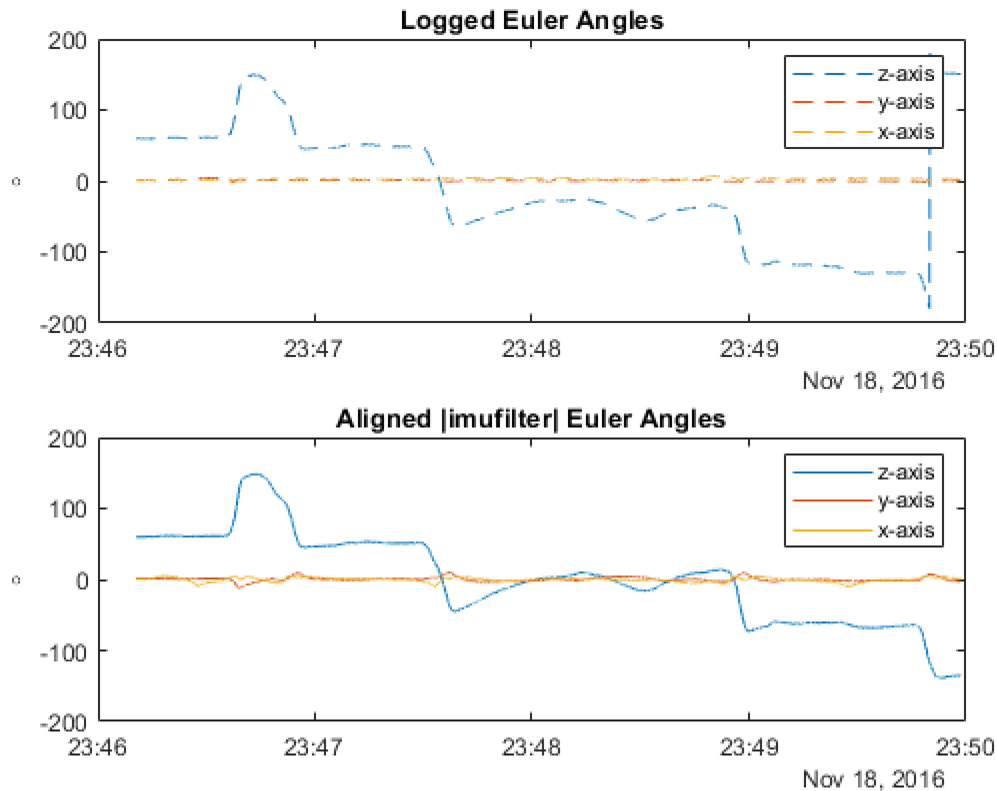


Align the Logged Orientation and `imufilter` Orientation

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.

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.

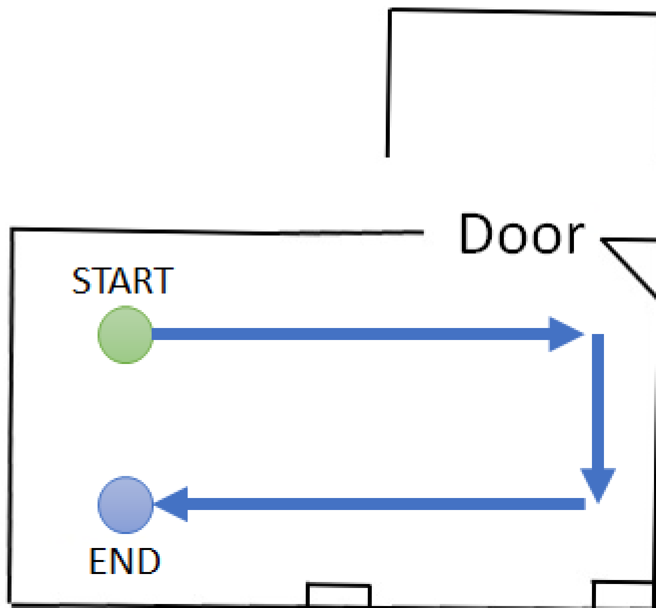
To estimate this pose, NDT subdivides the laser scan into 2D cells and each cell is assigned a corresponding normal distribution. The distribution represents the probability of measuring a point in that cell. Once the probability density is calculated, an optimization method finds the relative pose between 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.



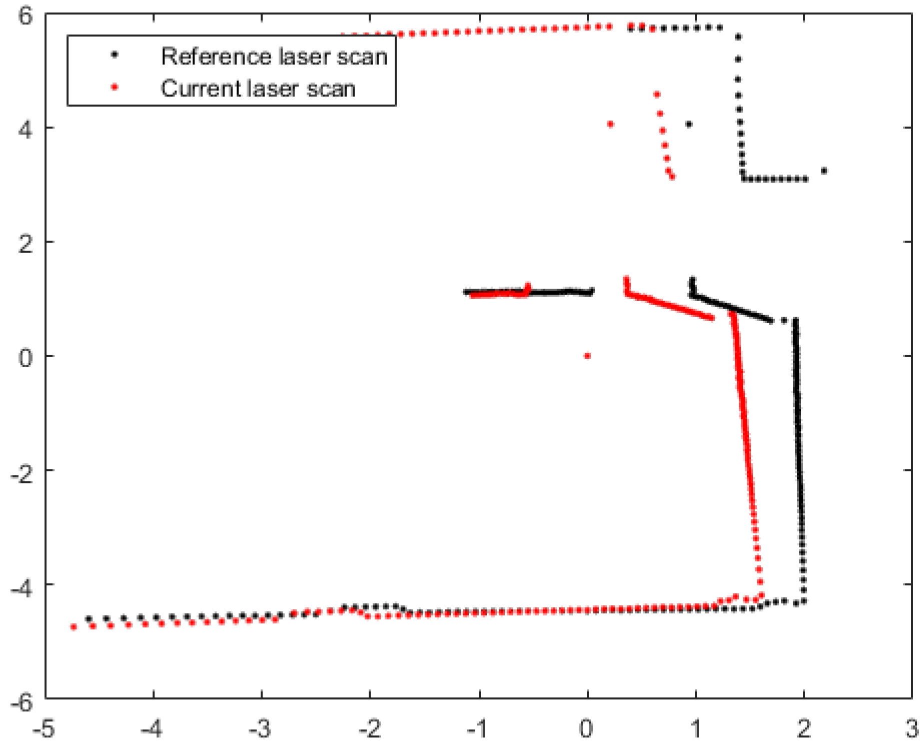
Plot Two Laser Scans

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)
```

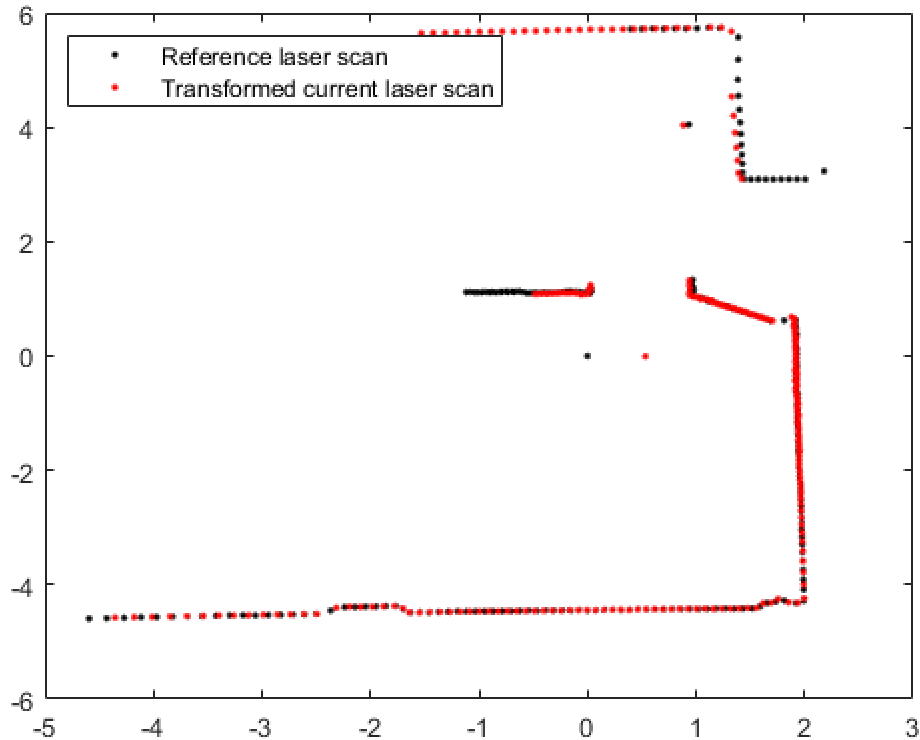
```
transform = 1×3

    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:
```

```
            XWorldLimits: [-7.5000 7.5000]
            YWorldLimits: [-7.5000 7.5000]
       OccupiedThreshold: 0.6500
           FreeThreshold: 0.2000
   ProbabilitySaturation: [0.0010 0.9990]
     GridLocationInWorld: [-7.5000 -7.5000]
                DataType: 'double'
            DefaultValue: 0.5000
              Resolution: 20
                GridSize: [300 300]
            XLocalLimits: [0 15]
            YLocalLimits: [0 15]
       GridOriginInLocal: [0 0]
      LocalOriginInWorld: [-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.

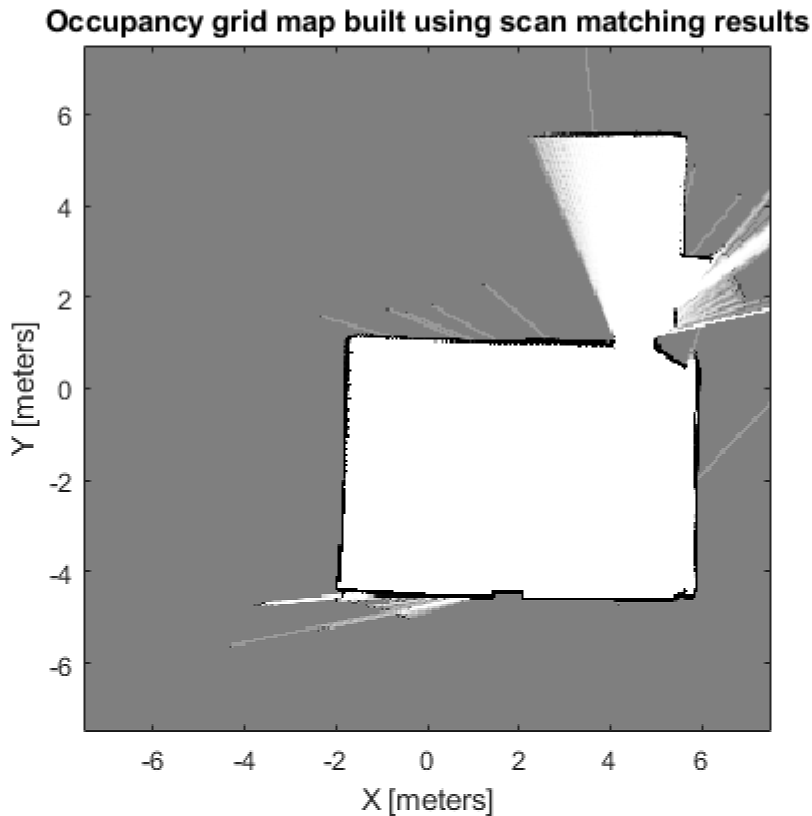
```
% Loop through all the scans and calculate the relative poses between them
for idx = 2:numScans
    % Process the data in pairs.
    referenceScan = lidarScans(idx-1);
    currentScan = lidarScans(idx);

    % 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
    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

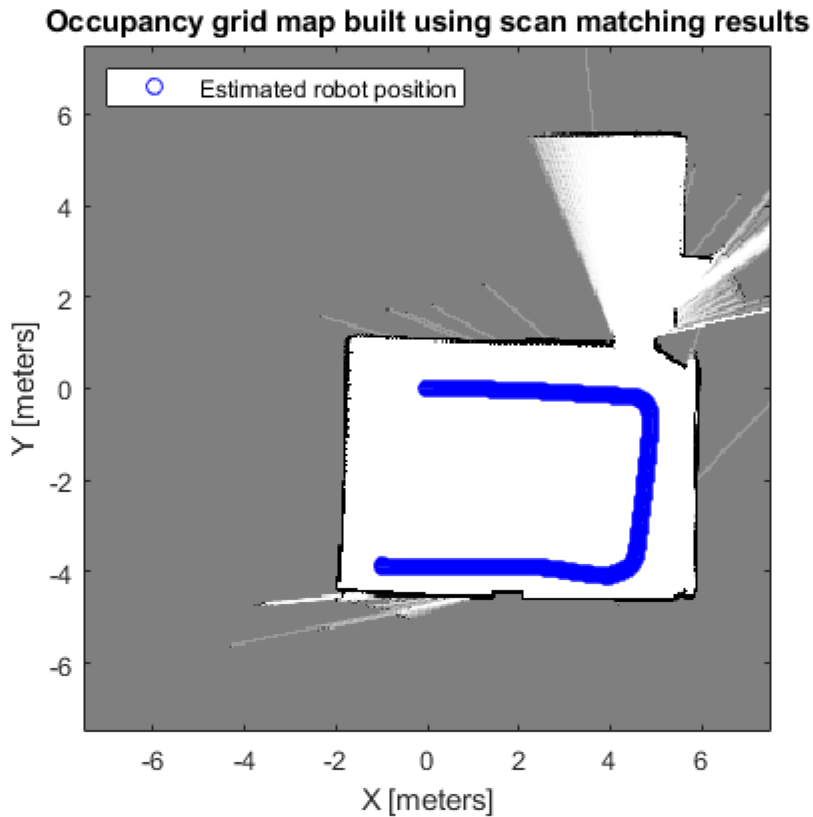
Visualize the occupancy grid map populated with the laser scans.

```
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 a 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

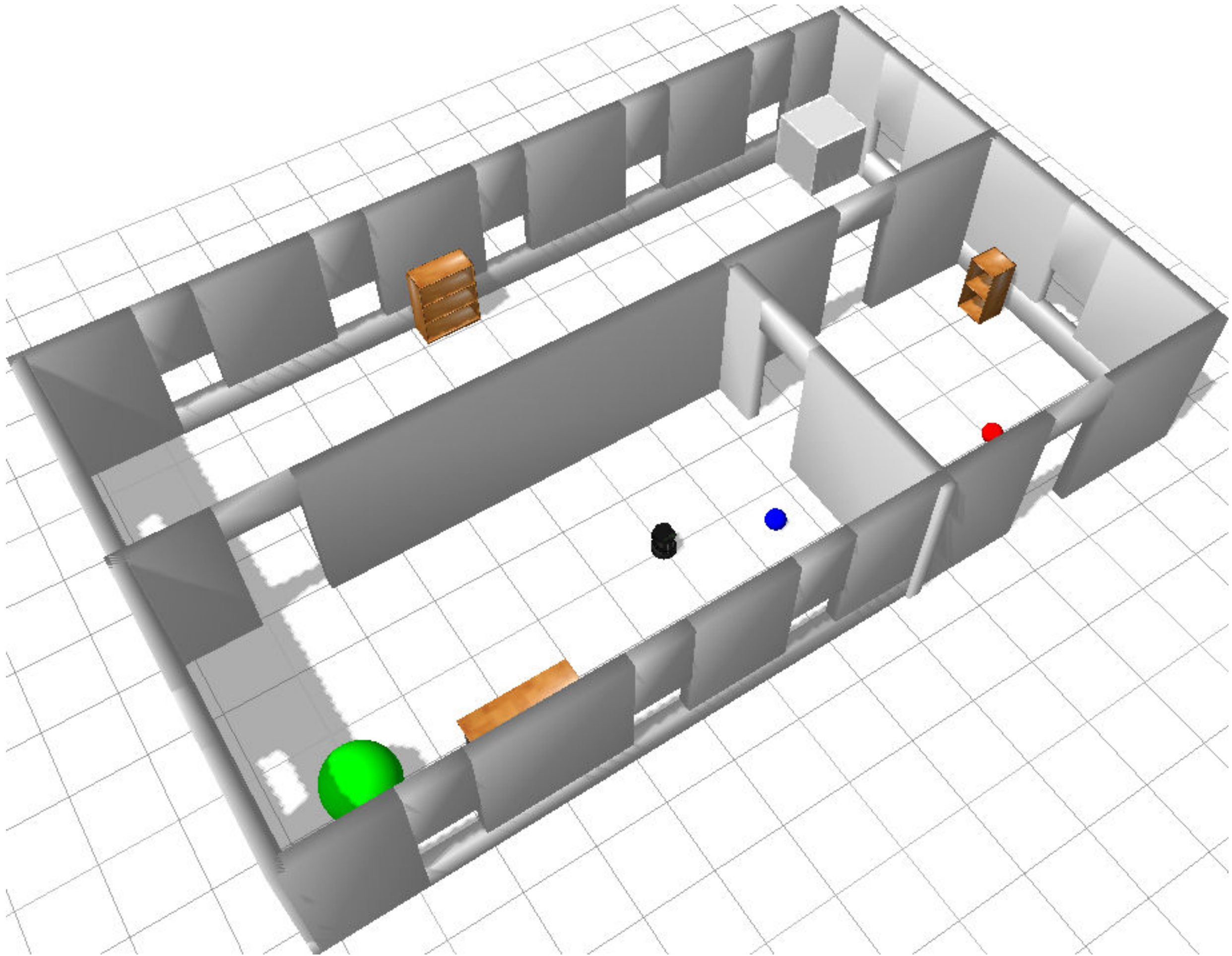
First, spawn a simulated TurtleBot inside an office environment in a virtual machine by following steps in the “Get Started with Gazebo and a Simulated TurtleBot” (ROS Toolbox) to launch the `Gazebo TurtleBot World` from the desktop.

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 = '192.168.233.133';
rosinit(ipaddress,11311);
```

```
Initializing global node /matlab_global_node_26847 with NodeURI http://192.168.233.1:64
```

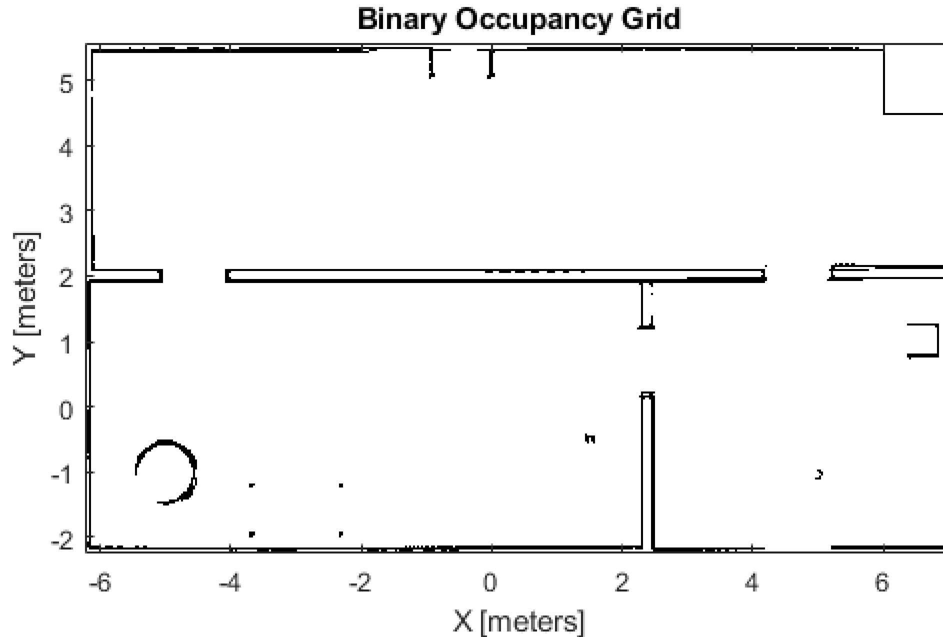
The layout of simulated office environment:



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 `likelihoodFieldSensor` 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','/camera_depth_frame');
sensorTransform = getTransform(tftree,'/base_link', '/camera_depth_frame');

% 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 la
```

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('/mobile_base/commands/velocity','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.

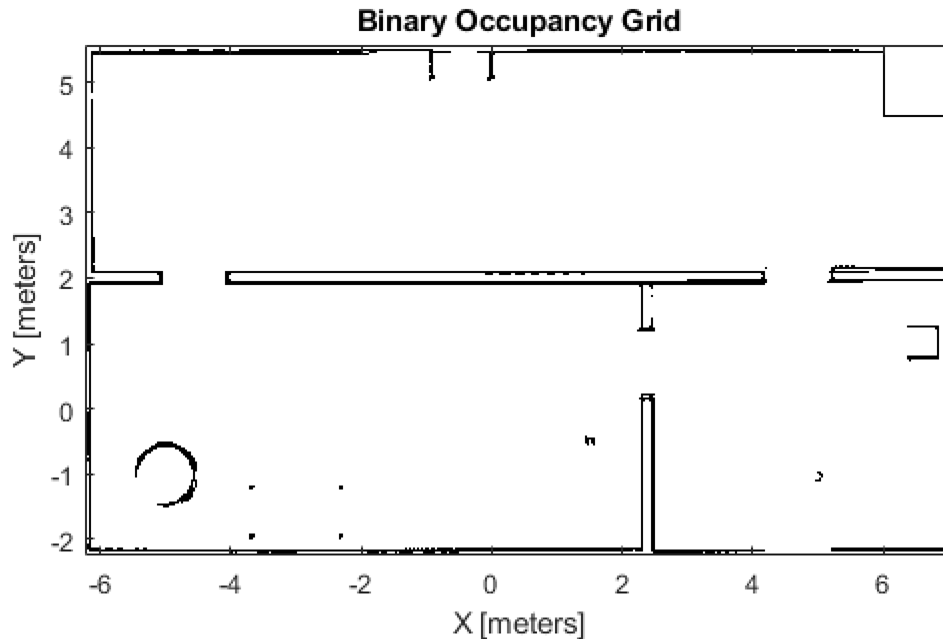
Please refer to section **Configure AMCL object for global localization** for an example on using global localization.

```
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);
```



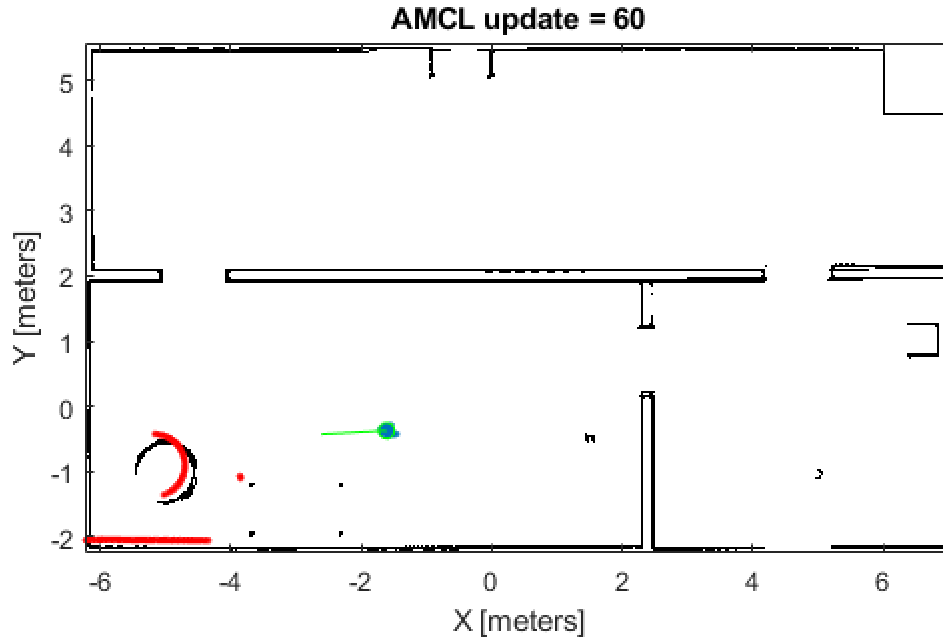
Robot motion is essential for the AMCL algorithm. In this example, we drive TurtleBot randomly using the ExampleHelperAMCLWanderer class, which drives the robot inside the environment while avoiding obstacles using the `controllerVFH` class.

```
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
```



Stop the TurtleBot and Shutdown ROS in MATLAB

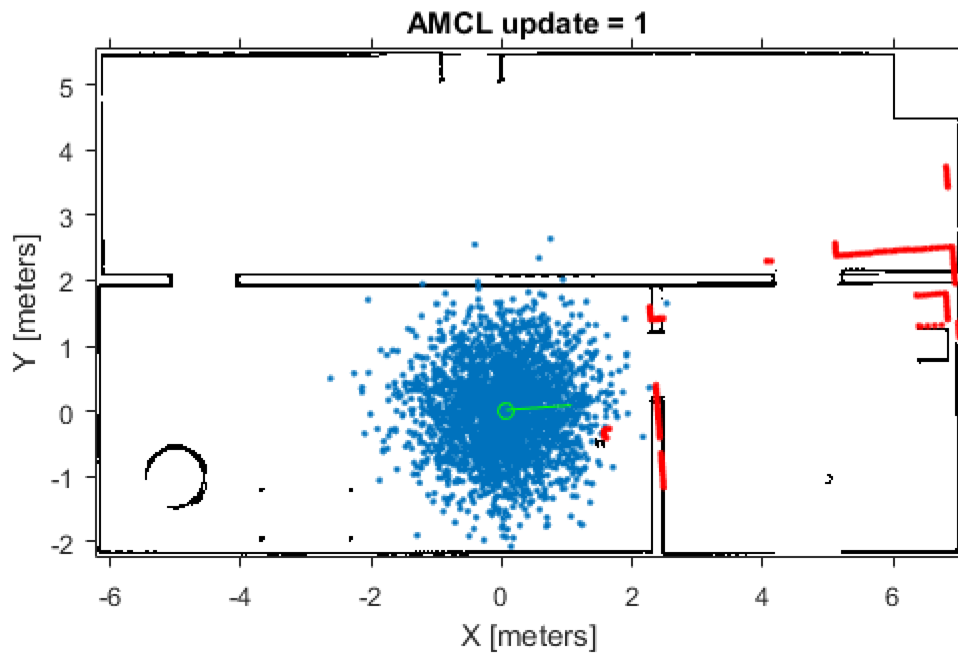
```
stop(wanderHelper);
rosshutdown
```

```
Shutting down global node /matlab_global_node_26847 with NodeURI http://192.168.233.1:6
```

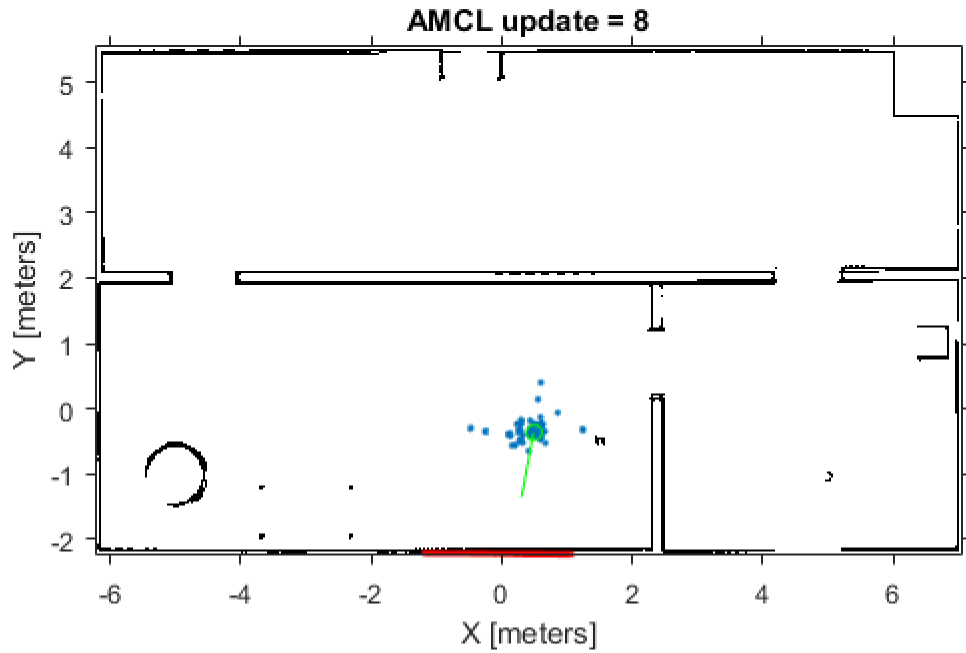
Sample Results for AMCL Localization with Initial Pose Estimate

AMCL is a probabilistic algorithm, the simulation result on your computer may be slightly different from the sample run shown here.

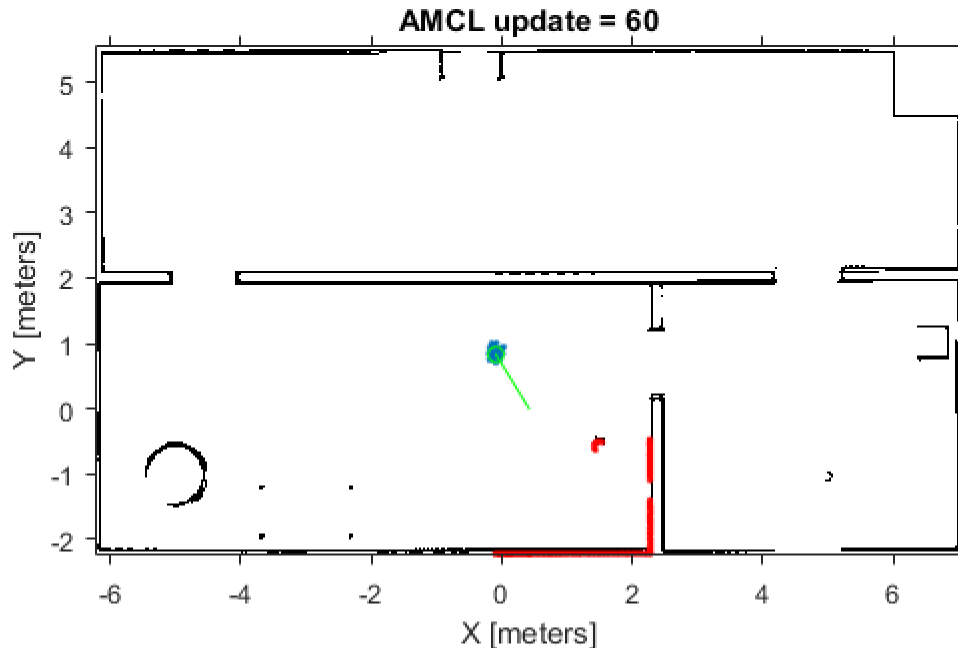
After first AMCL update, particles are generated by sampling Gaussian distribution with mean equal to `amcl.InitialPose` and covariance equal to `amcl.InitialCovariance`.



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.

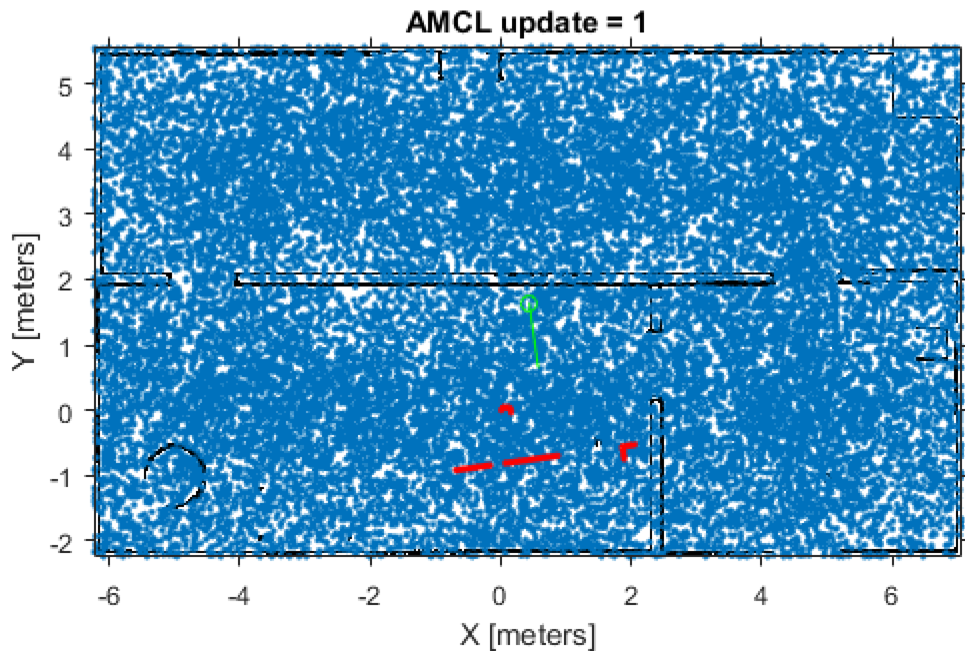
To enable AMCL global localization feature, replace the code sections in **Configure AMCL object for localization with initial pose estimate** with the code in this section.

```
amcl.GlobalLocalization = true;
amcl.ParticleLimits = [500 50000];
```

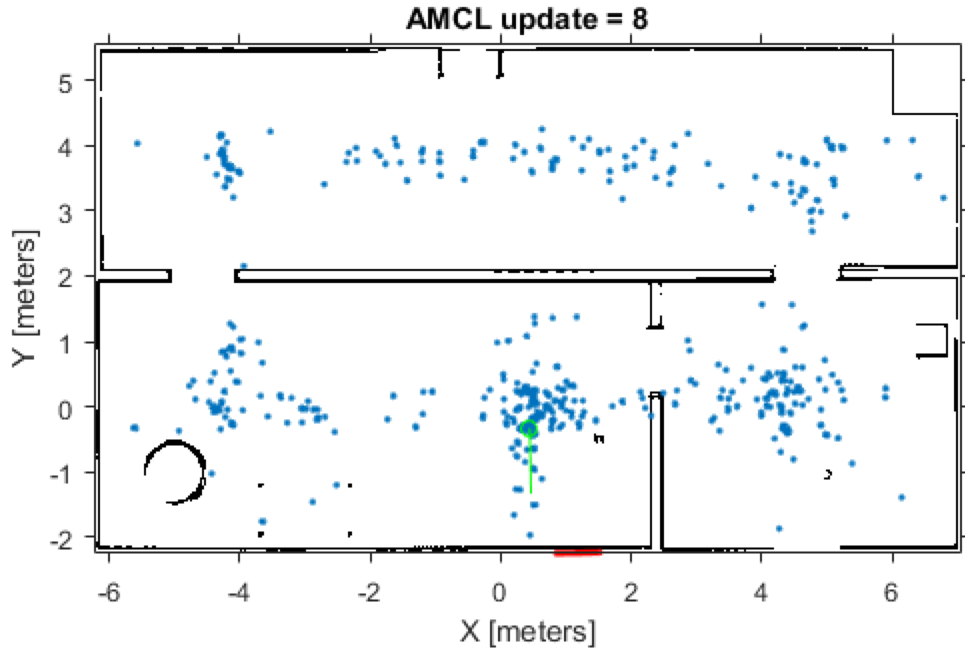
Sample Results for AMCL Global Localization

AMCL is a probabilistic algorithm, the simulation result on your computer may be slightly different from the sample run shown here.

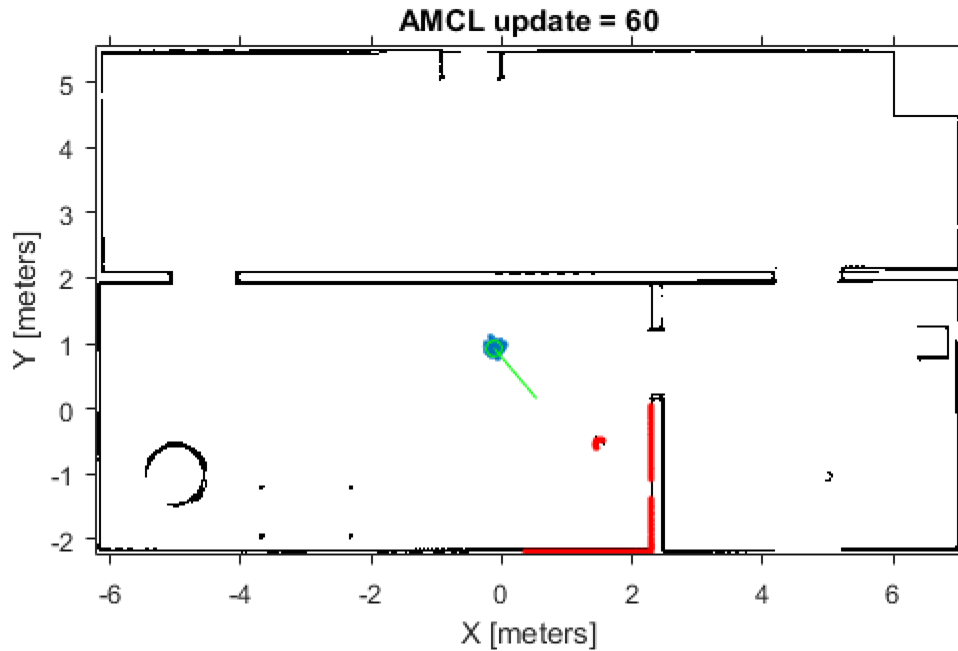
After first AMCL update, particles are uniformly distributed inside the free office space:



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

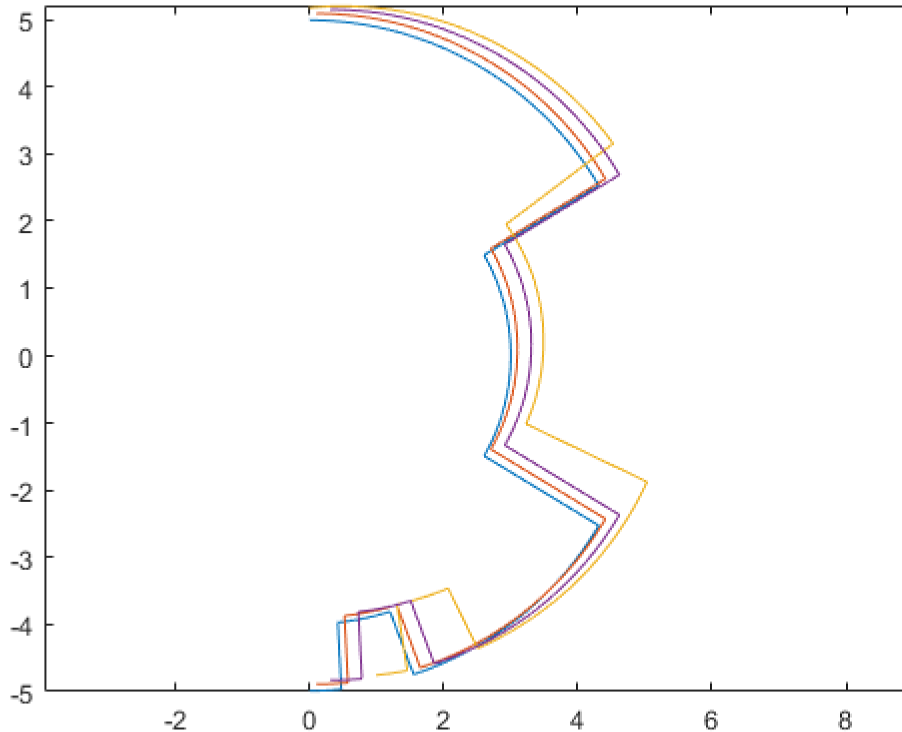
[1] S. Thrun, W. Burgard and D. Fox, Probabilistic Robotics. Cambridge, MA: MIT Press, 2005.

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]`.

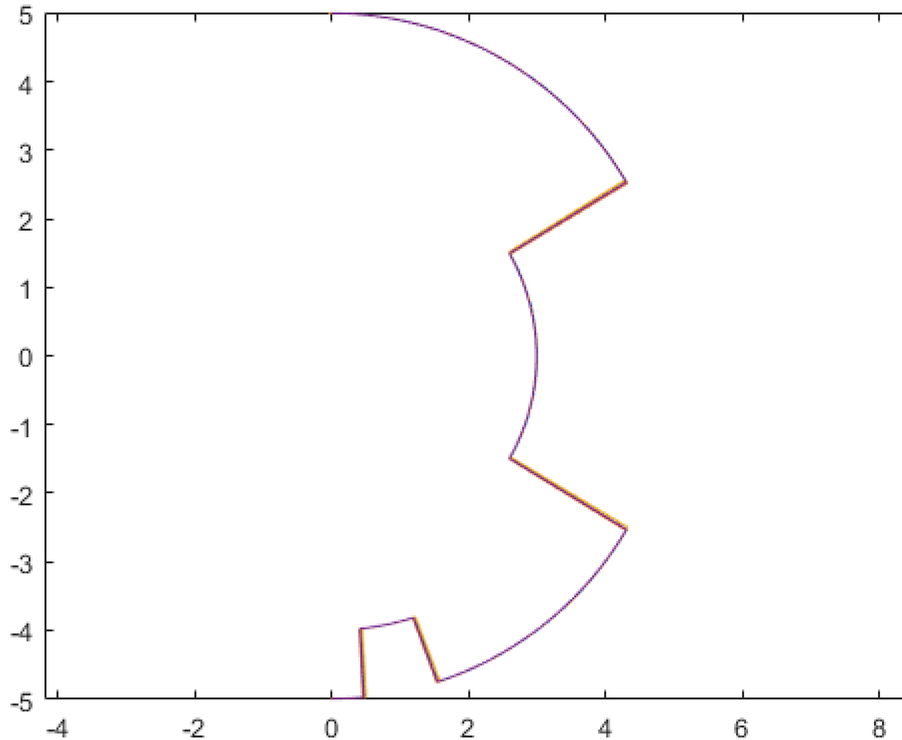
```
relPoses(1,:) = [0 0 0];

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),compose
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

matchScans | transformScan

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;
```

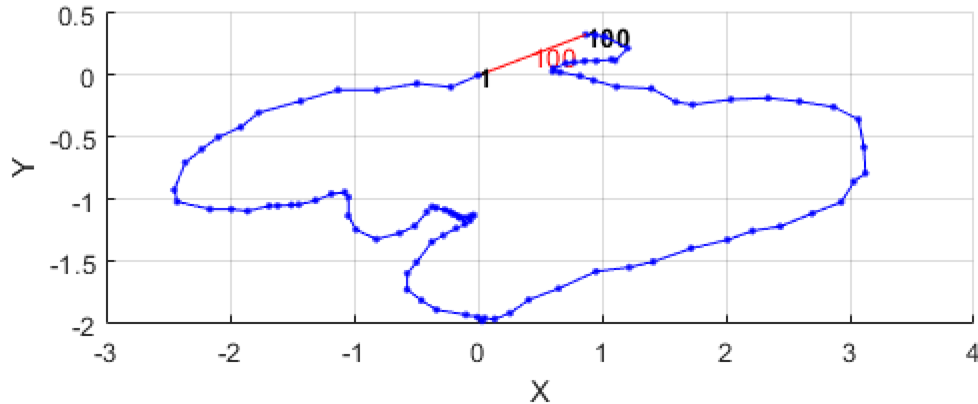
Add nodes to the pose graph, with edges defining the relative pose and information matrix for the pose graph. Convert the estimated poses, given as transformations, to relative poses as an `[x y theta qw qx qy qz]` vector. An identity matrix is used for the information matrix for each pose.

```
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.

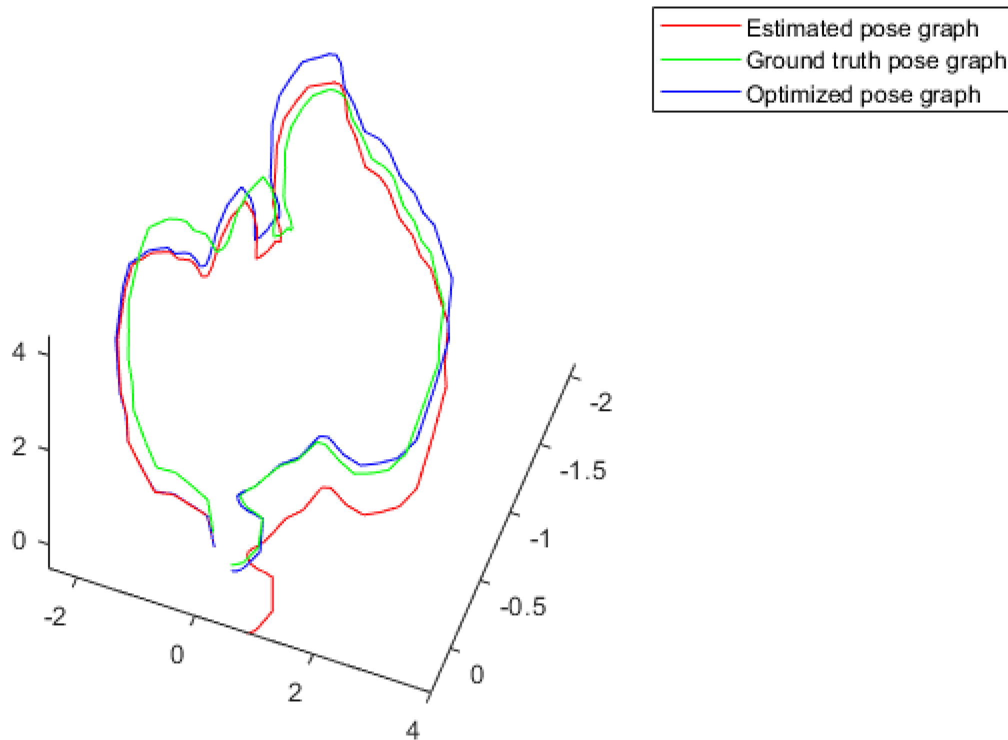
```
% Convert pose from transformation to pose vector.
relativeQuat = tform2quat(loopedge);
relativePose = [tform2trvec(loopedge),relativeQuat];
% Loop candidate
loopcandidateframeid = 1;
% Current frame
currentframeid = 100;

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 probabilitistic information, where 0 indicates free-space and 1 idicates 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 and 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, relative to the world origin

```
worldMap.LocalOriginInWorld = mapData.GridLocationInWorld;
```

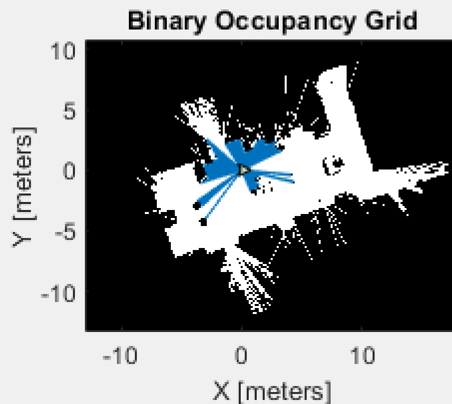
Plot the ground truth. This example sets up a subplot for showing two maps side by side.

```
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 to 0. 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. Offset the grid-origin relative to the local-origin by setting the `GridOriginInLocal` property to half the difference between the world limits. This shifts the bounds of the map so that the local-origin is centered.

```
% 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+
```

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;

% 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 mononically increasing array of timestamps, tStamps.

```
% Get distance between each waypoint
pt2ptDist = distance(stateSpace,path.States(1:end-1,:),path.States(2:end,:))
```

```
pt2ptDist = 139×1

    0.1818
    0.1818
    0.1818
    0.1818
    0.1818
    0.1818
    0.1818
    0.1818
    0.1818
    0.1818
      ⋮
```

```
linVel = 0.5; % m/s
tStamps = cumsum(pt2ptDist)/linVel;
```

Generate a final simulated trajectory with `waypointTrajectory,` specifying the path states, the corresponding timestamps, and a desired sample rate of 10Hz.

```
traj = waypointTrajectory(path.States, [0; tStamps], 'SampleRate', 10);
```

Simulate Sensor Readings and Build a Map

Lastly, move the robot along the trajectory while updating the map with the simulated Lidar readings.

To initialize the simulation, reset the trajectory, set the local origin to the first xy point on the trajectory, and clear the map.

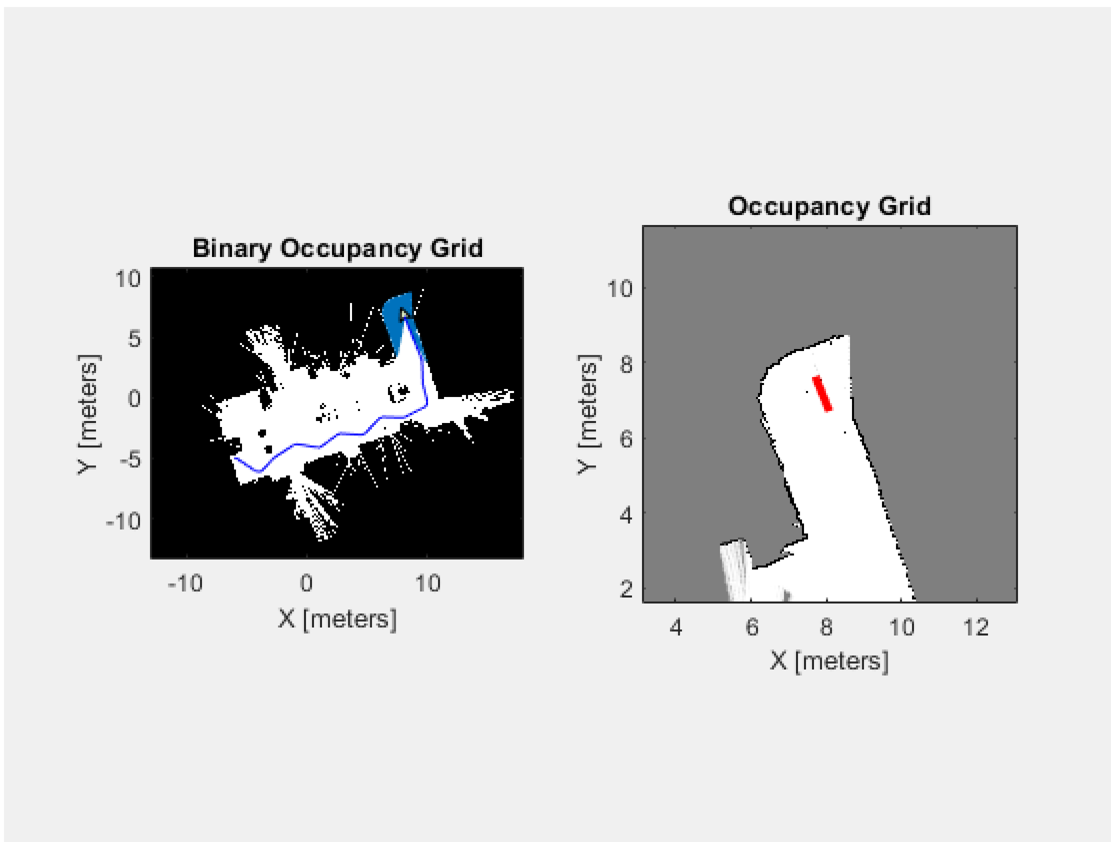
```
reset(traj);
robotCurrentPose = path.States(1,:);
move(egoMap, robotCurrentPose(1:2));
setOccupancy(egoMap,repmat(egoMap.DefaultValue,egoMap.GridSize));
```

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 lidar.
- Update the local map with sensor data using `insertRay`.
- Refresh the visualization.

```
while ~isDone(traj)
    % Increment robot along trajectory
    [pts, quat] = step(traj);

    % Get orientation angle from quaternion
    rotMatrix = rotmat(quat,'point');
    orientZ = rotm2eul(rotMatrix);

    % Move the robot to the new location
    robotCurrentPose = [pts(1:2) orientZ(1)];
    move(egoMap, robotCurrentPose(1:2),'MoveType','Absolute');

    % 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
```

```
    % Update world plot
    exampleHelperUpdateRobotAndLidar(hRobot, hSensorData, robotCurrentPose, ranges, ang

    % Call drawnow to push updates to the figure
    drawnow limitrate
end
```



Note the robot drives through the environment, simulating sensor readings and building an occupancy map as it goes.

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`.

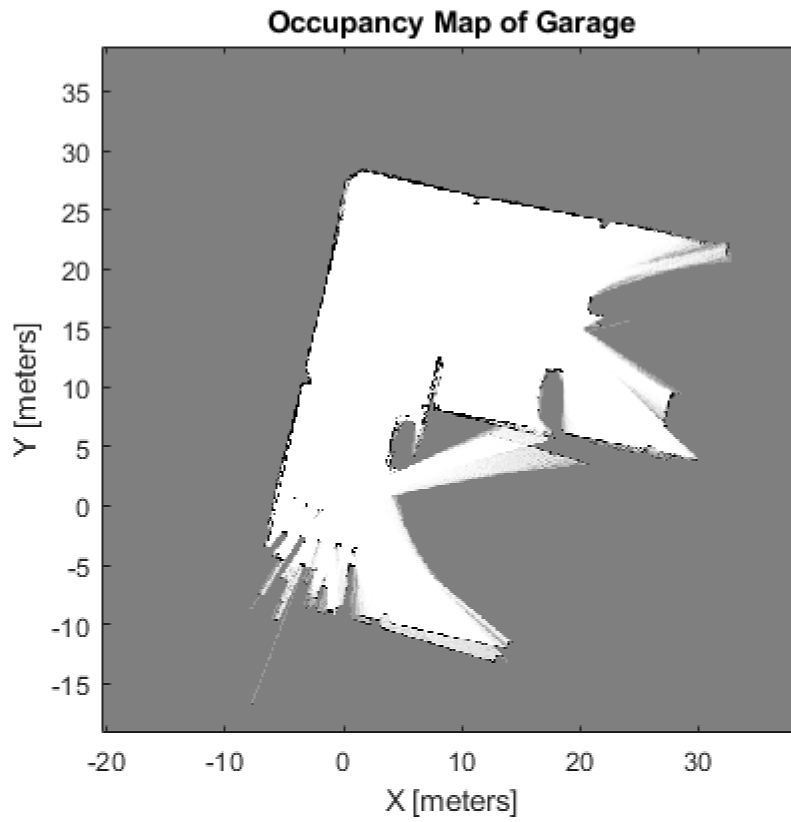
Load scan and pose estimates collected from sensors on a robot in a parking garage. The data collected is correlated using a `lidarSLAM` algorithm, which performs scan matching to associate scans and adjust poses over the full robot trajectory. Check to make sure scans and poses are the same length.

```
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')
```


Occupancy Map of Garage
35
30
25
20
15
10
5
0
-5
-10
-15
Y [meters]
-20
-10
0
10
20
30
X [meters]

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 heterogenous sensors into a common temporal frame of reference. This is usually done using an egocentric occupancy map. This map contains information of 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.

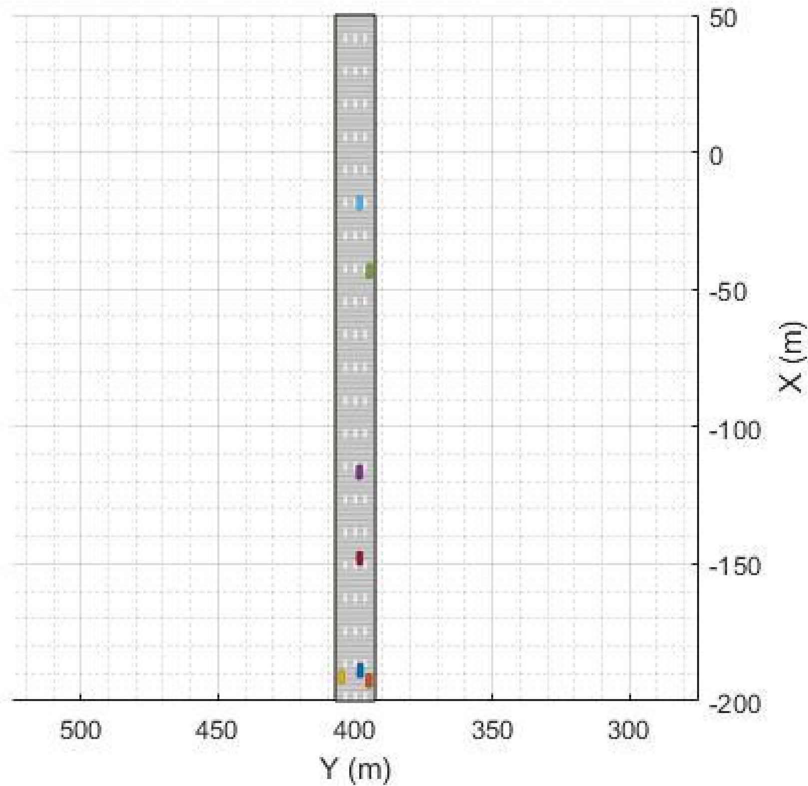
This example also uses a straight road scenario designed using Driving Scenario Designer (DSD). For more details see Driving Scenario Designer. User can also create a Driving Scenario programmatically. For more details please refer to “Driving Scenario Tutorial” (Automated Driving Toolbox).

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 vision detection generator which synthesizes camera sensor mounted at the front of the ego vehicle in driving scenario. This generator is configured to detect lane boundaries and obstacles.

For more details, see `visionDetectionGenerator`.

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 `exampleHelperGetObstacleDataFromSensors`.

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 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 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.

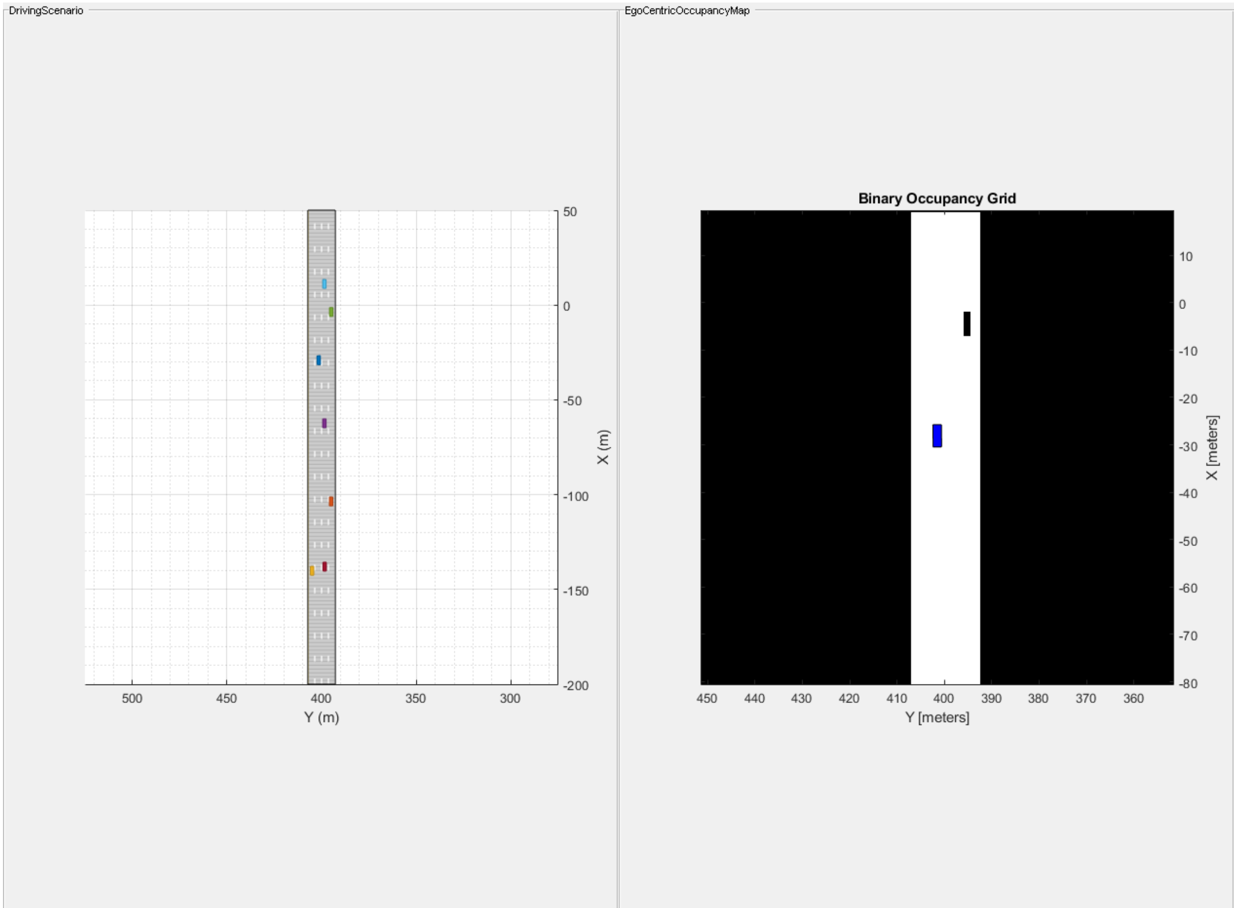
```
    [obstacleInfo, roadBorders, isValidLaneTime] = ...
      exampleHelperGetObstacleDataFromSensor(scenario, egoMap, ...
                                 egoVehicle, sensor);
```

Depending upon the range of the vision detection generator, the detections may fall outside the `egoMap` bounds. Using the `egoMap` bounds, the `exampleHelperFitlerObstacles` 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
```


DrivingScenario
EgoCentricOccupancyMap
Binary Occupancy Grid
Y (m)
X (m)
Y [meters]
X [meters]

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;
```

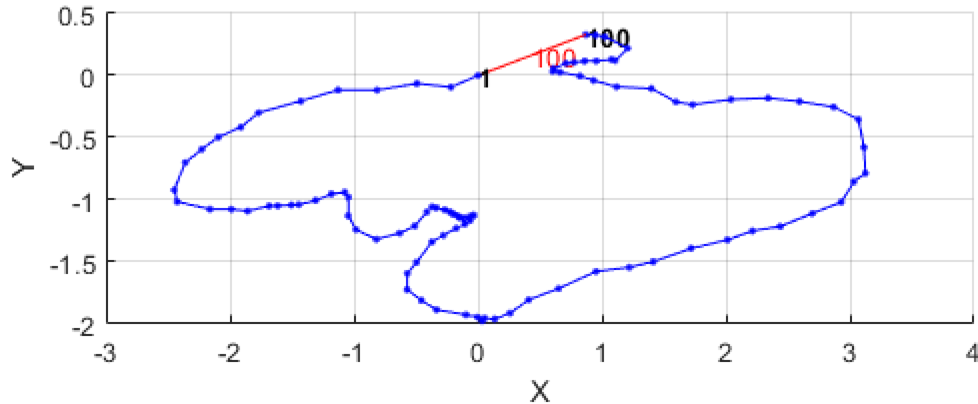
Add nodes to the pose graph, with edges defining the relative pose and information matrix for the pose graph. Convert the estimated poses, given as transformations, to relative poses as an `[x y theta qw qx qy qz]` vector. An identity matrix is used for the information matrix for each pose.

```
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. 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].

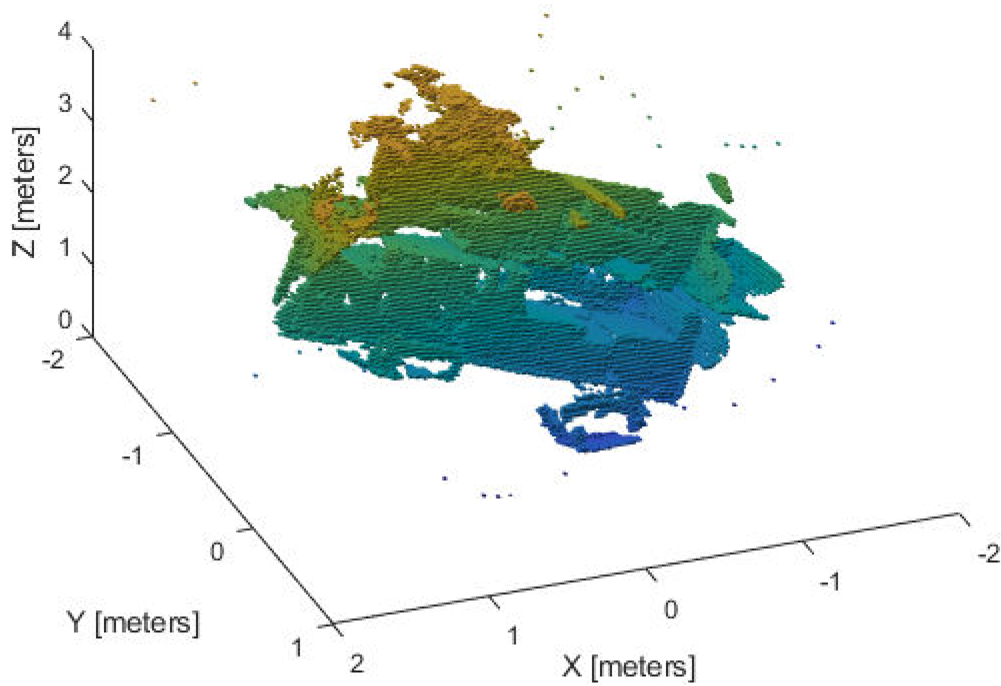
```
load('depthimagearray.mat');    % variable depthImages
load('freburgK.mat');           % variable K
```

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.

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

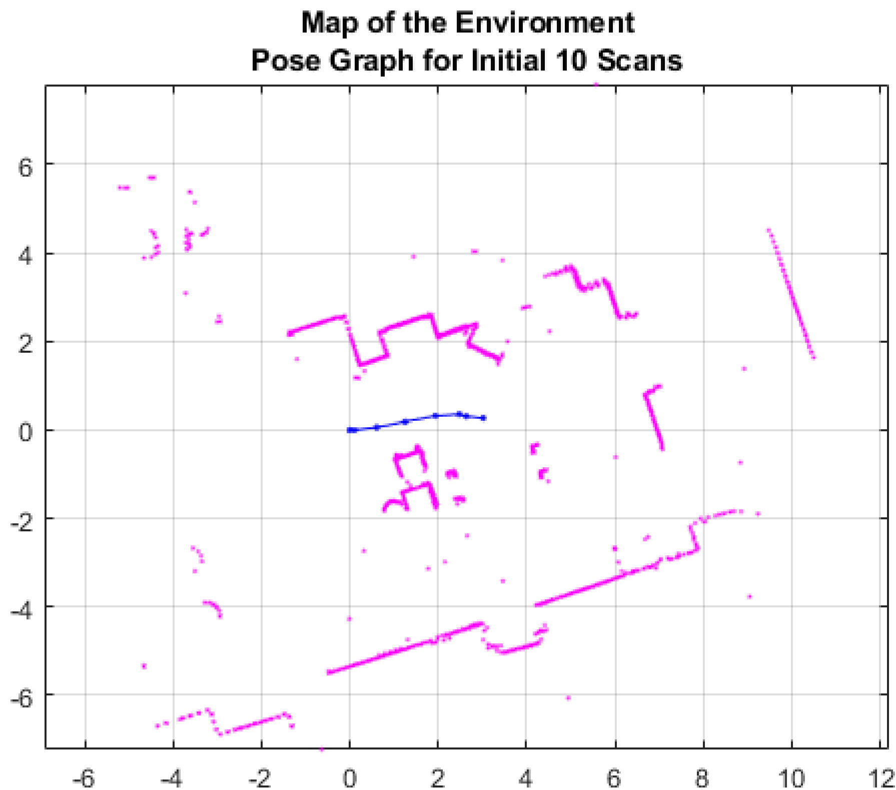
Incrementally add scans to the `slamAlg` object. Scan numbers are printed if added to the map. The object rejects scans if the distance between scans is too small. Add the first 10 scans first to test your algorithm.

```
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'});
```



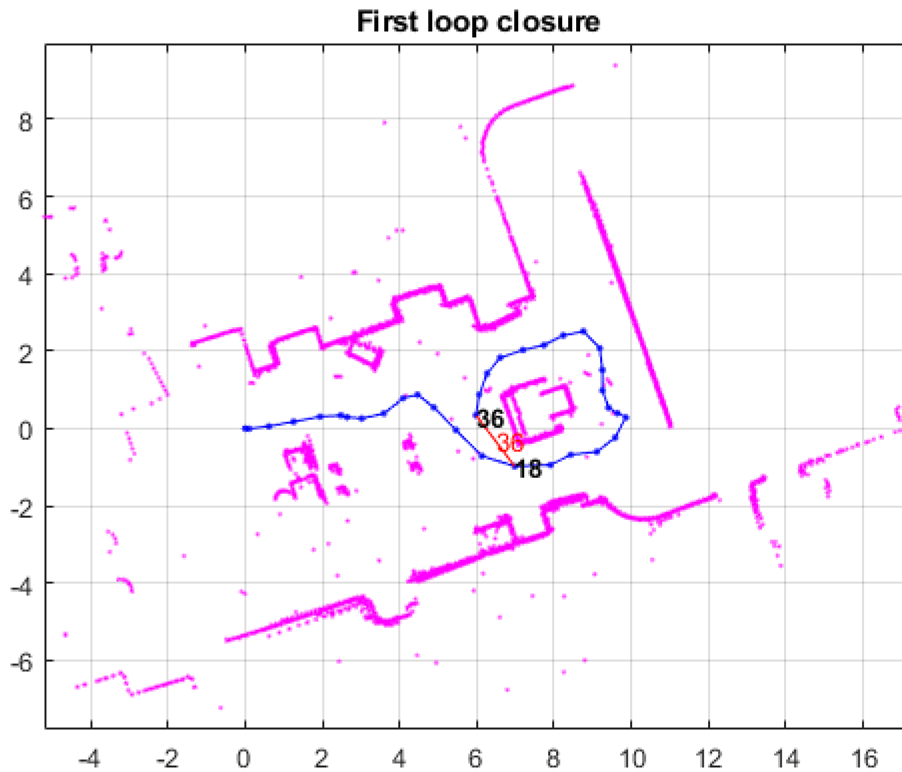
Observe the Effect of Loop Closures and the Optimization Process

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;
```

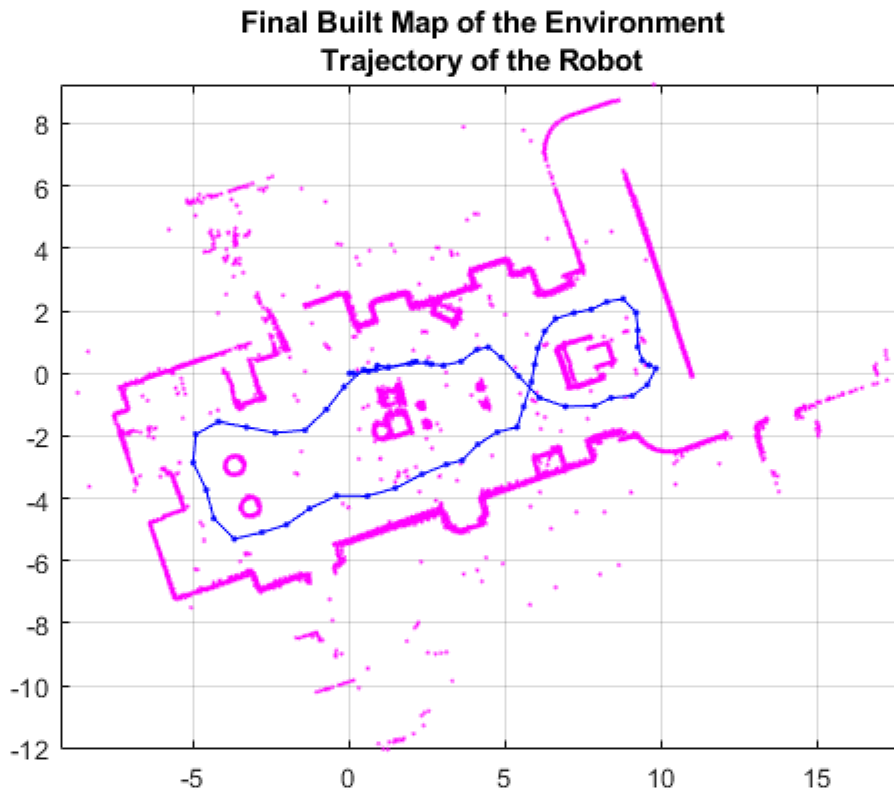
```
for i=10:length(scans)
    [isScanAccepted, loopClosureInfo, optimizationInfo] = addScan(slamAlg, scans{i});
    if ~isScanAccepted
        continue;
    end
    % visualize the first detected loop closure, if you want to see the
    % complete map building process, remove the if condition below
    if optimizationInfo.IsPerformed && ~firstTimeLCDetected
        show(slamAlg, 'Poses', 'off');
        hold on;
        show(slamAlg.PoseGraph);
        hold off;
        firstTimeLCDetected = true;
        drawnow
    end
end
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. The previous `for` loop should have added all the scans despite only plotting the initial loop closure.

```
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.

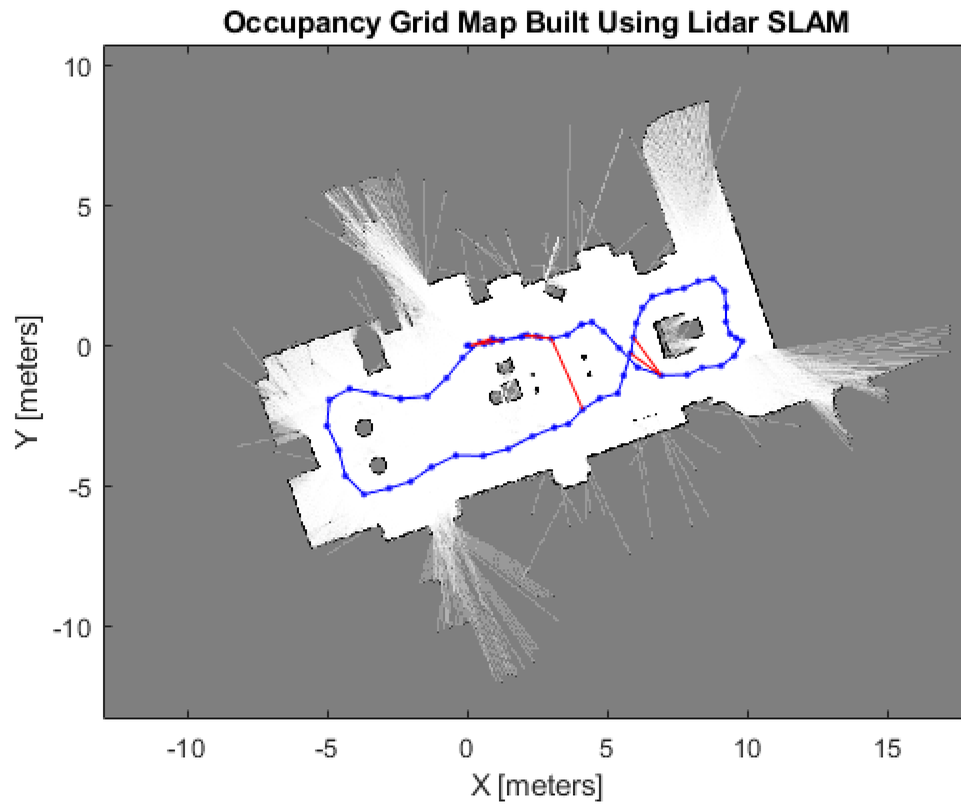
Build Occupancy Grid Map

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.

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 1-181 example. This example requires Simulink® 3D Animation™ and Robotics System Toolbox™.

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

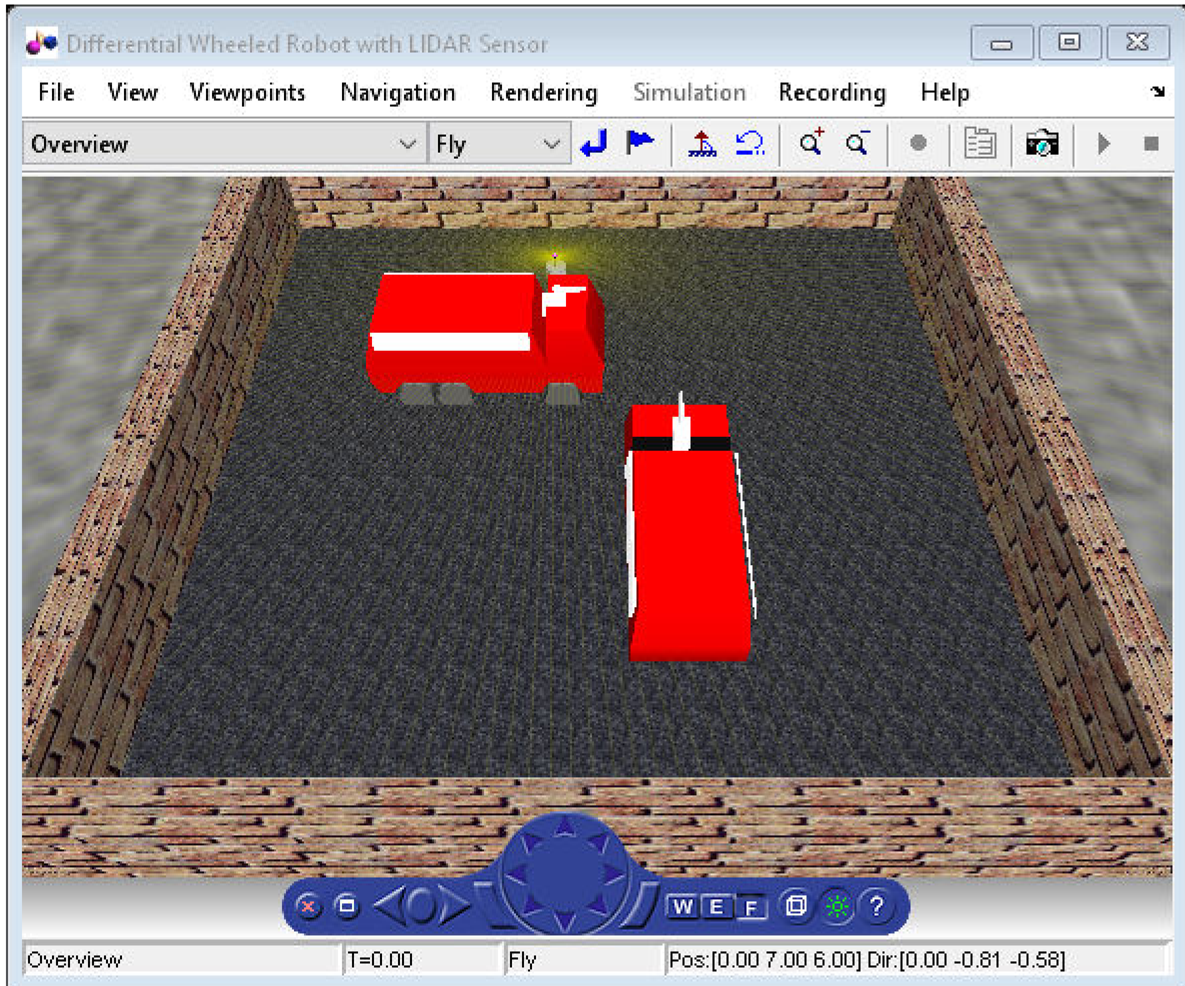
This example uses a virtual scene with two vehicles and four walls as obstacles and a robot equipped with a lidar scanner shown in the Simulink 3D Animation Viewer. You can navigate in a virtual scene using the menu bar, toolbar, navigation panel, mouse, and keyboard. Key features of the viewer are illustrated in the `VR 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 = vrnode(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 vrnode.

```
lidarVRNode = vrnode(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`.

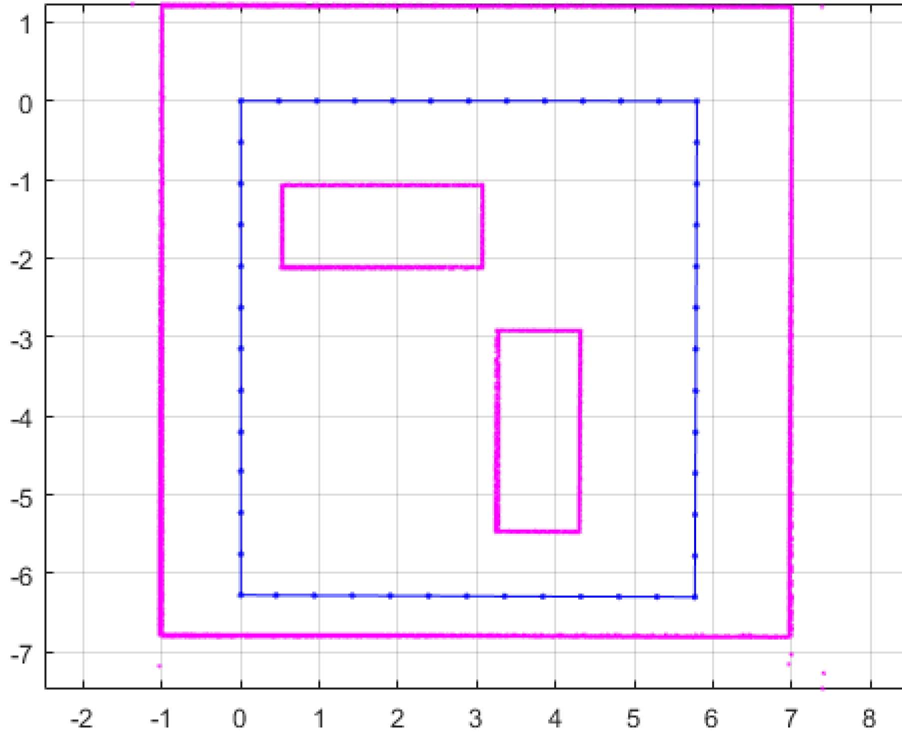
A snapshot is shown to demonstrate of the scans and poses when the first loop closure is identified and verify the results visually. This plot shows overlaid scans and an optimized pose graph for the first loop closure.

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

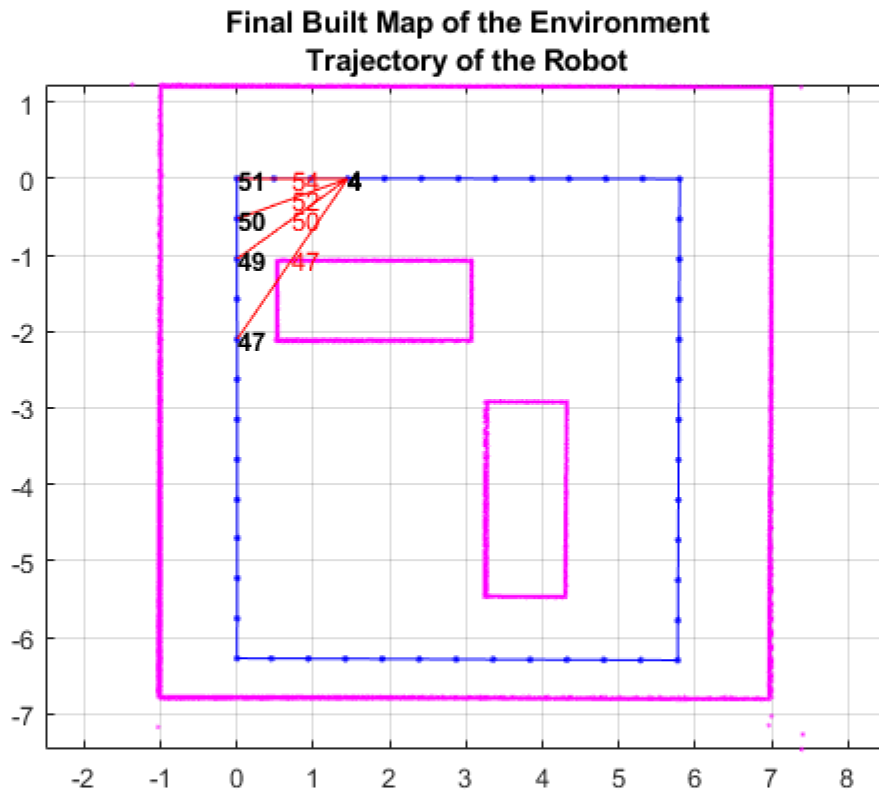
```
firstLoopClosure = false;
scans = cell(length(trajectory),1);

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
```



Plot the final built map after all scans are added to the `slamAlg` object.

```
show(slamAlg,'Poses','off');
hold on
show(slamAlg.PoseGraph);
hold off
title({'Final Built Map of the Environment','Trajectory of the Robot'});
```



Build Occupancy Grid Map

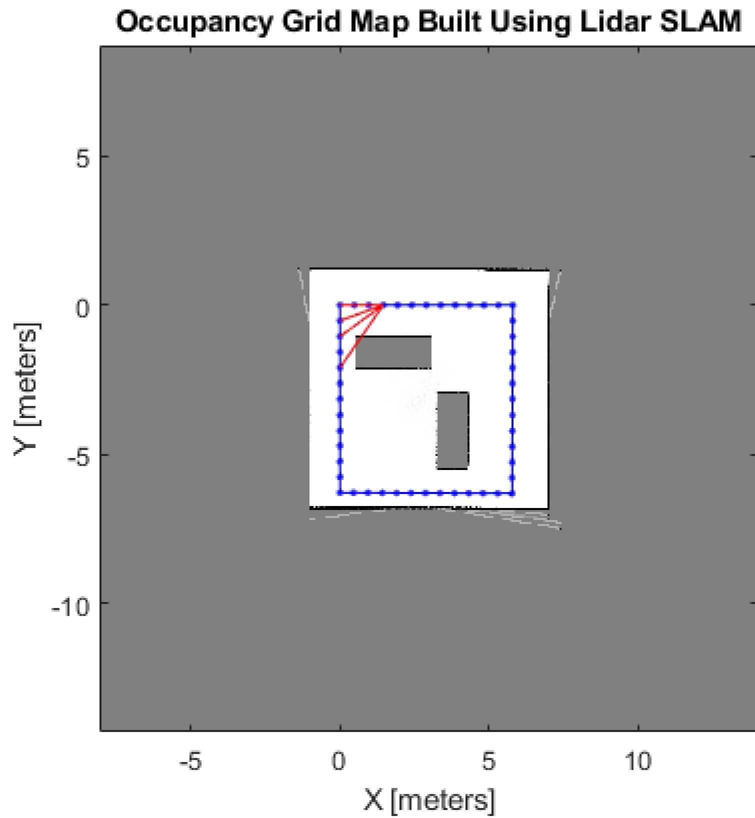
The optimized scans and poses can be used to generate a `occupancyMap` which represents the environment as a probabilistic occupancy grid.

```
[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 3 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 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. So 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 in 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 closure is searched only 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. 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 to remove the remaining points on floor and ceiling after the region of interest extraction using point cloud plane fit algorithms.

```
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;
```

Initilize 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 = [1,0,0,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1];
% Number of loop closure edges added since last pose graph optimization and map refiner
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.

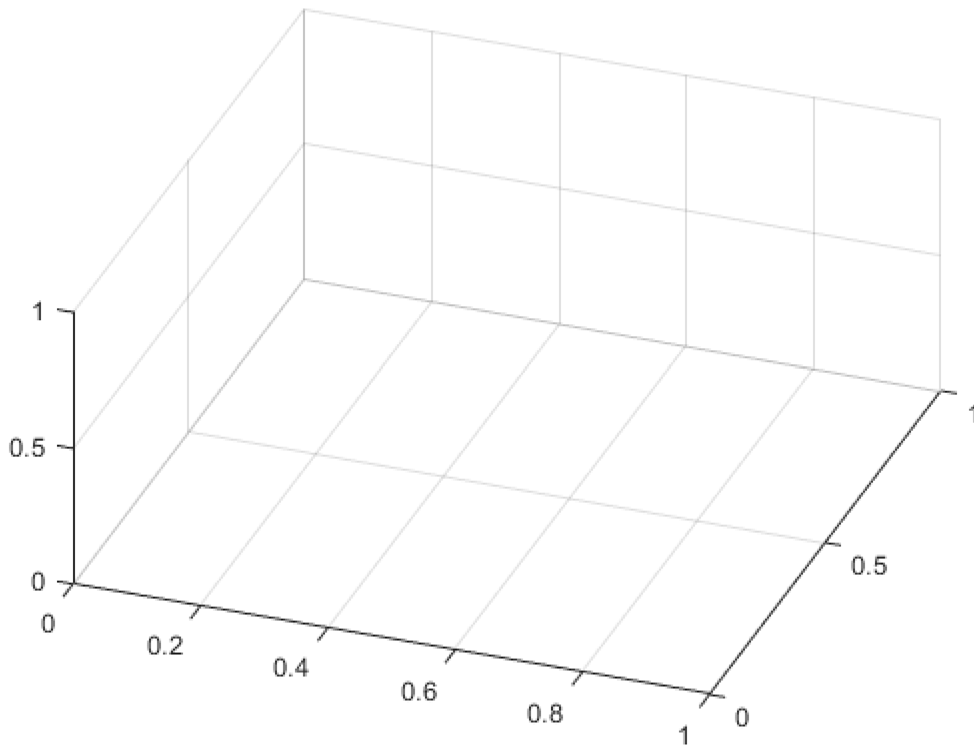
```
% Set to 1 to visualize created map and posegraph during build process
viewMap = 1;
% Set to 1 to visualize processed point clouds during build process
viewPC = 0;
```

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] = ...
        pcfitplane(pcl, maxDistance,referenceVector,maxAngularDistance);
```

```
pcl_wogrd = select(pcl,outliers,'OutputSize','full');
```

Remove points on the ceiling plane.

```
[~, ~, outliers] = ...
    pcfitplane(pcl_wogrd,0.2,referenceVector,maxAngularDistance);
pcl_wogrd = select(pcl_wogrd,outliers,'OutputSize','full');
```

Select points in annular region.

```
ind = (pcl_wogrd.Location(:,3)<annularRegionLimits(2))&(pcl_wogrd.Location(:,3)>ann
pcl_wogrd = select(pcl_wogrd,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_wogrd_sampled = pcdownsample(pcl_wogrd,'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 can
    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
```

Loop Closure Query

Loop closure query happens after accepting a scan. The query involves finding a similarity between the current scan and previously accepted scans. A similar scan found is called a loop candidate. A loop candidate estimation consists of the following steps

- Estimate submaps matching the current scan are estimated using submap and scan matching algorithm.
- Each submap represents `nScansPerSubmap` consecutive scans. The previous scans represented by all matching submaps are considered as matching with the current scan (probable loop candidates).
- The relative pose between the current scan and probable loop candidate is computed using NDT registration algorithm. Loop candidate with the least relative pose estimation error (RMSE) is considered as the best match for the current scan.
- A best match is accepted as a valid loop closure only when the RMSE is less than a specified threshold.

```
    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] = pcregisterndt(pcl_wogrd_sampled,...
                        pcProcessed{probableLoopCandidate},gridStep);
                    % Update best Loop Closure Candidate
                    if rmse < rmseMin
                        loopCandidate = probableLoopCandidate;
                        rmseMin = rmse;
                    end
                    if rmseMin < rmseThreshold
                        break;
                    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 = numLoopClosuresSinceLastOpti
            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 uncertanty is pose estimation reduces after pose graph 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}.removeInvalidPoint
            end
            mapUpdated = true;
            ax = newplot;
            grid on;
        end
        numLoopClosuresSinceLastOptimization = 0;
        % Reduce the frequency of optimization after optimizing
        % the trjectory
        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.

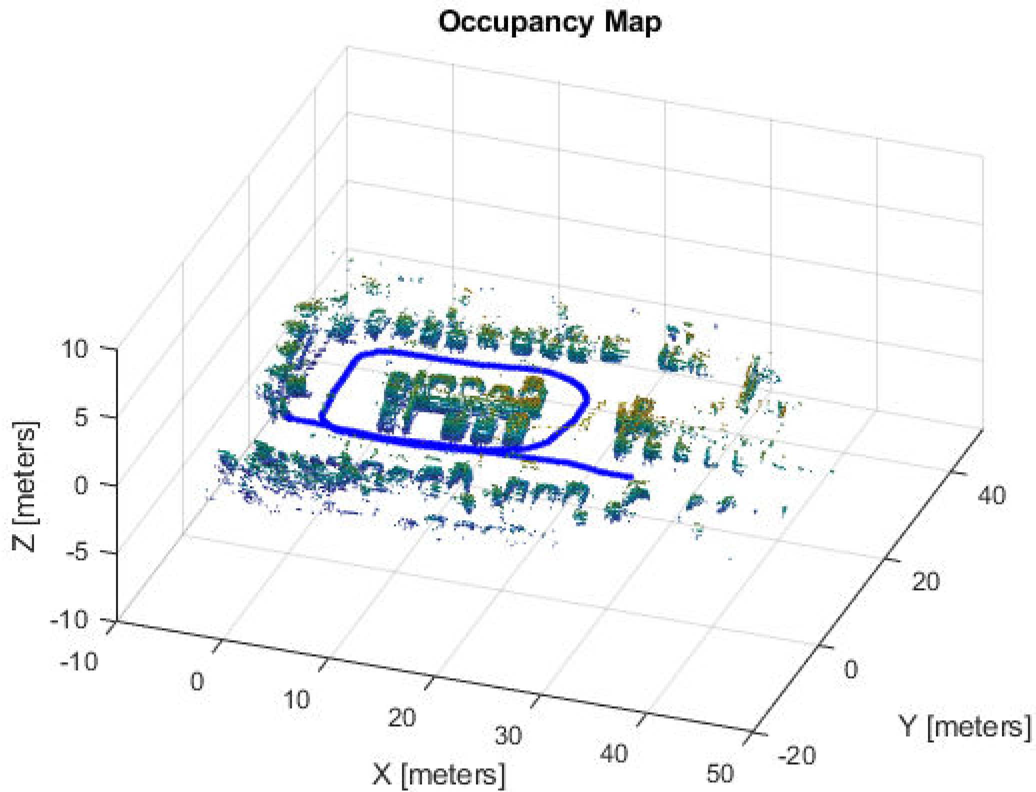
```
    pcToView = pcdownsample(pcl_wogrd_sampled, 'random', 0.5);
    pcsToView{count} = pcToView;

    if viewMap==1
        % Insert point cloud to the occupance 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
```

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
```

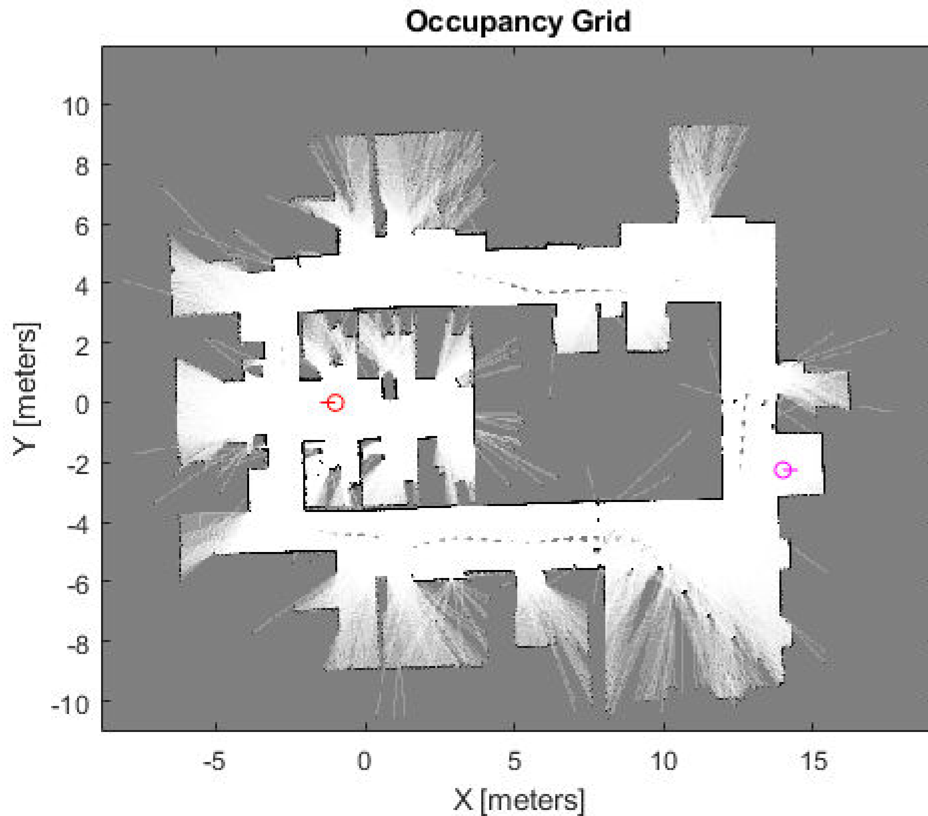
Plan Mobile Robot Paths using RRT

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 the start and goal poses
start = [-1.0, 0.0, -pi];
goal = [14, -2.25, 0];

% Show the start and goal positions of the robot
hold on
plot(start(1), start(2), 'ro')
plot(goal(1), goal(2), 'mo')

% Show the 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 0.4m 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 `ValdiationDistance` of 0.05m. This 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
```

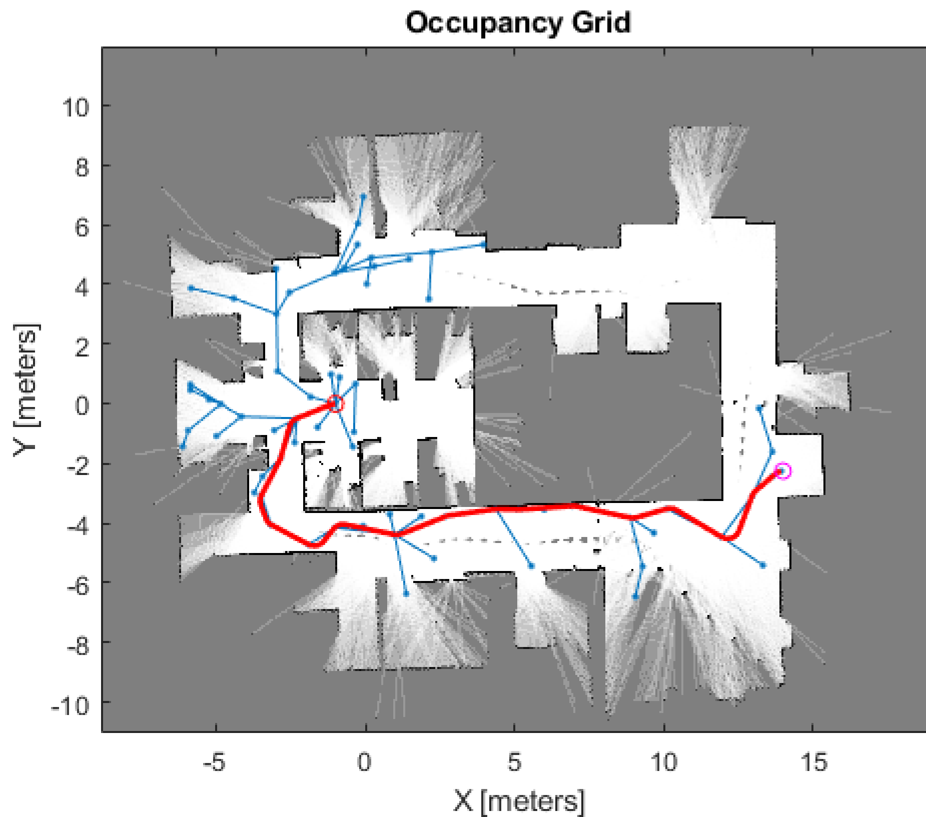
Plan the path between the start and goal. Because of the random sampling, this example sets the `rng` seed for consistent results.

```
rng(0,'twister')

[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

% 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 the start and goal in the 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 uses (see `dubinsConnection.DisabledPathTypes`).

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 The 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;
```

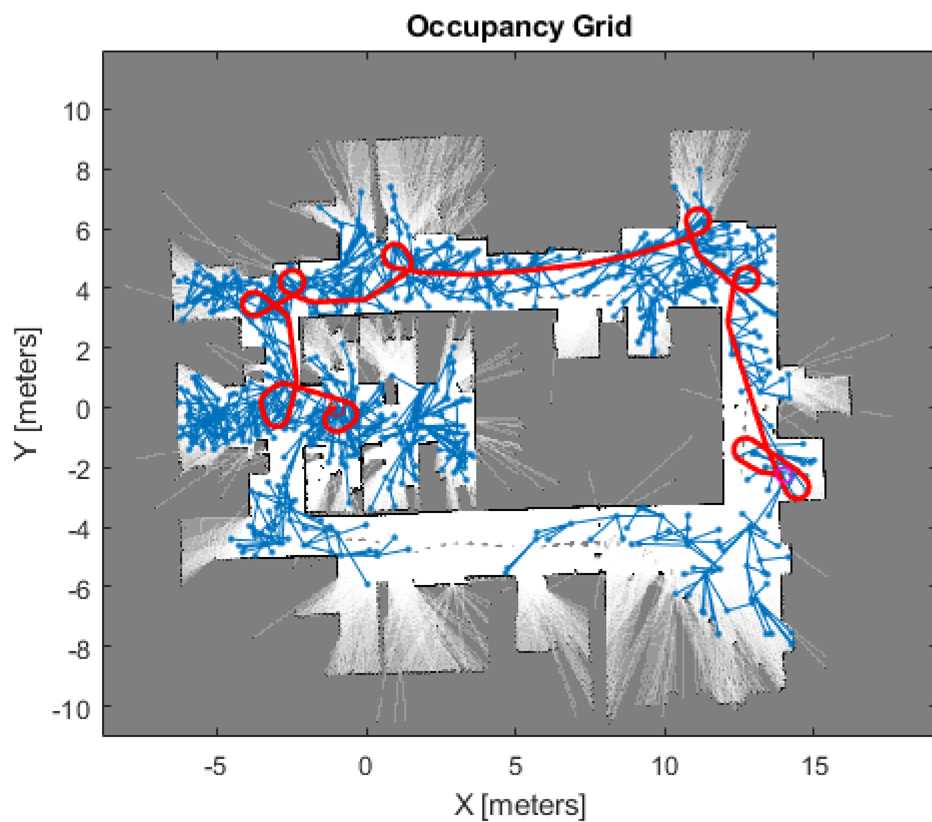
Plan the path between the start and goal. Reset the `rng` seed again.

```
rng(0, 'twister')
[pthObj2, solnInfo] = plan(planner, start, goal);
```

Plot the path

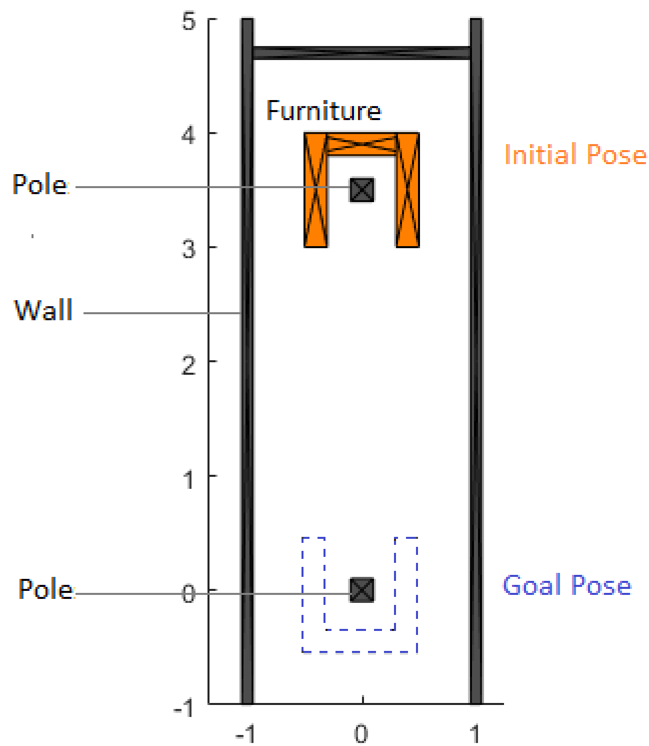
Draw the new path on the map. The path should only execute left turns to reach the goal. This example shows how you can customize your constraints and still plan paths using the generic RRT algorithm.

```
figure
show(occGrid)

hold on

% show the search tree
plot(solnInfo.TreeData(:,1), solnInfo.TreeData(:,2), '.-'); % tree expansion

% draw path (after the path is interpolated)
pthObj2.interpolate(300)
plot(pthObj2.States(:,1), pthObj2.States(:,2), 'r-', 'LineWidth', 2)

% Show the start and goal in the grid map
plot(start(1), start(2), 'ro')
plot(goal(1), goal(2), 'mo')
hold off
```


Occupancy Grid
10
8
6
4
2
0
-2
-4
-6
-8
-10
Y [meters]
-5
0
5
10
15
X [meters]

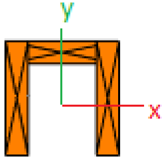
Moving Furniture in a Cluttered Room with RRT

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*) algoirthm. Provided example helpers illustrate how to define custom state spaces and state valdiation for any motion planning application.



Model the Scene

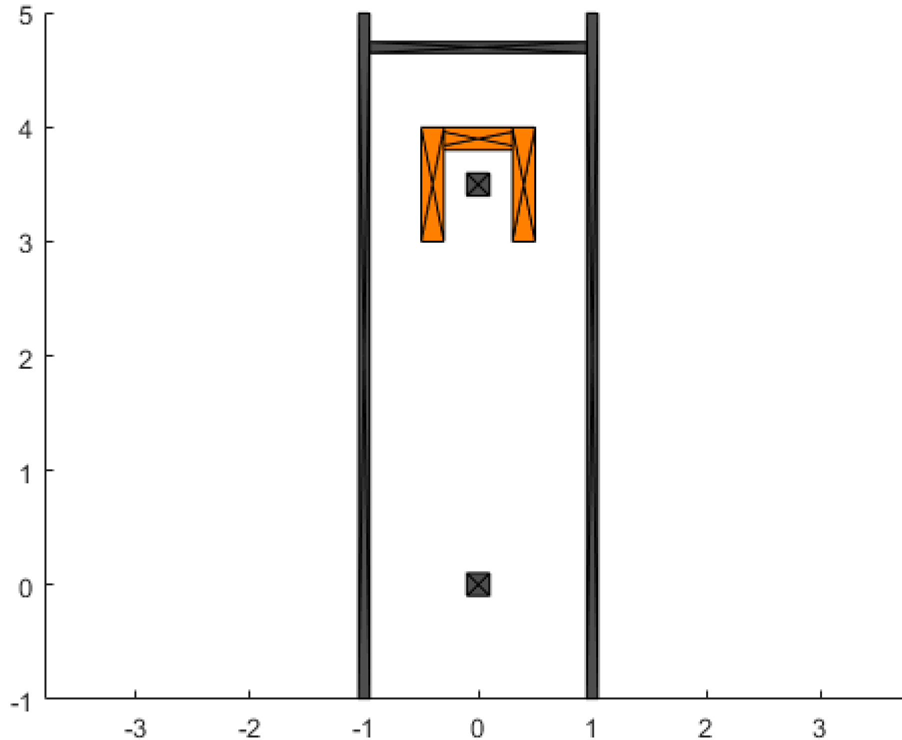
To help visualize and solve this path planning problem, two helper classes are provided, `ExampleHelperFurniture` and `ExampleHelperRoom`. The `ExampleHelperFurniture` class assembles the furniture by putting three rectangle collision boxes together. The furniture is set up as below:



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 `statestateSpaceSE2` 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

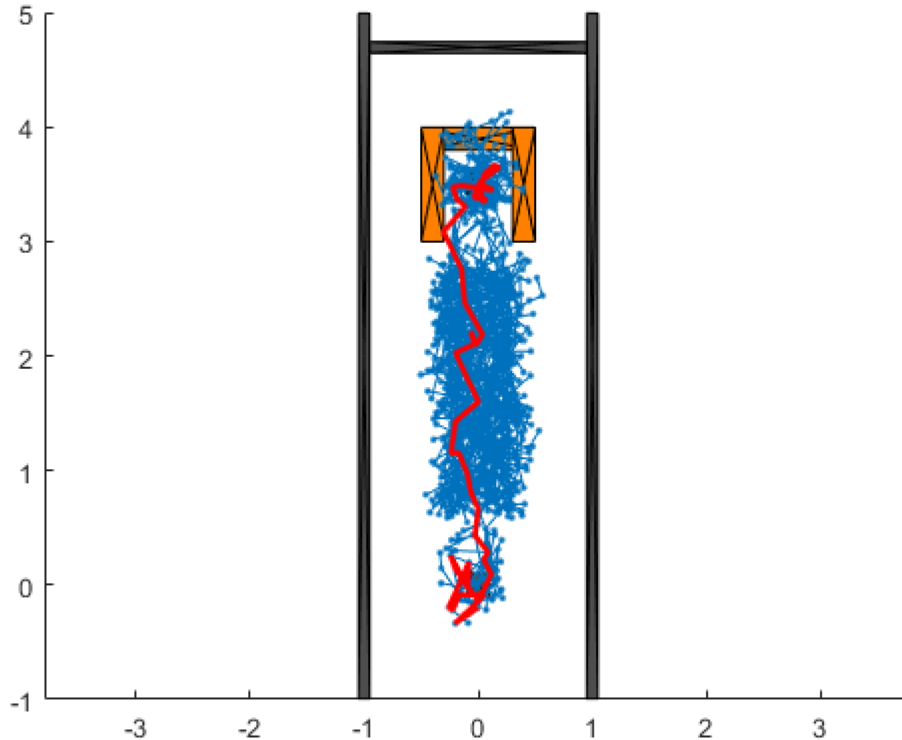
Use `plannerRRTStar` as the planner and specify the custom state space and state validator. Specify additional parameters for the planner. The `GoalReached` example helper function returns `true` when a feasible path gets close enough to the goal within a threshold. This exits the planner.

```
% 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 `pi` 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), '.-');
% Interpolate path and plot points
interpolate(path,300)
plot(path.States(:,1), path.States(:,2), 'r-', 'LineWidth', 2)

hold off
```

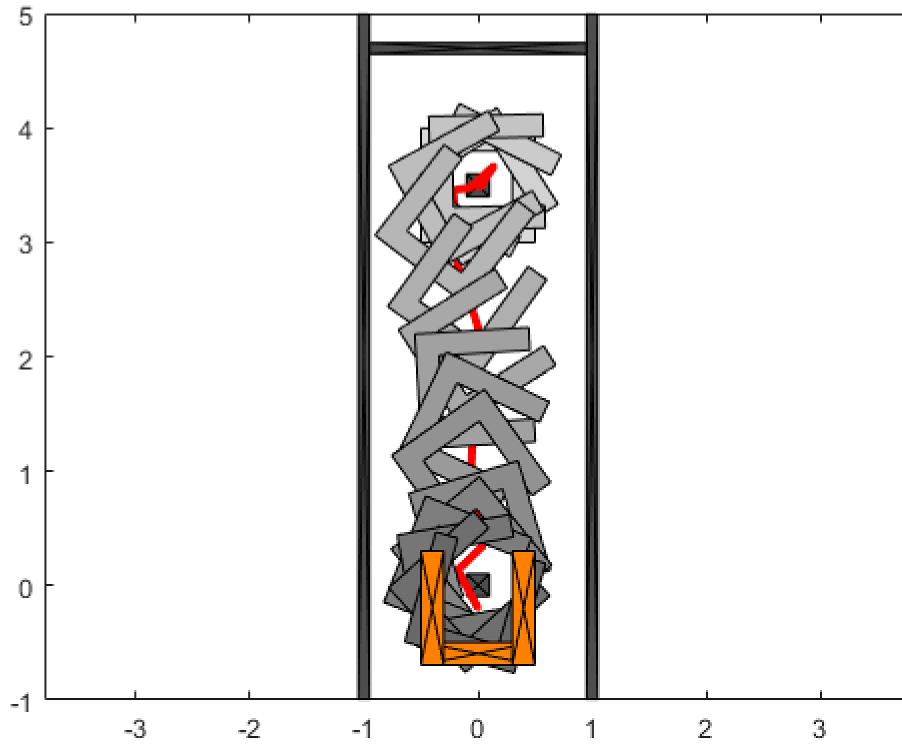


Visualize the Motion

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.

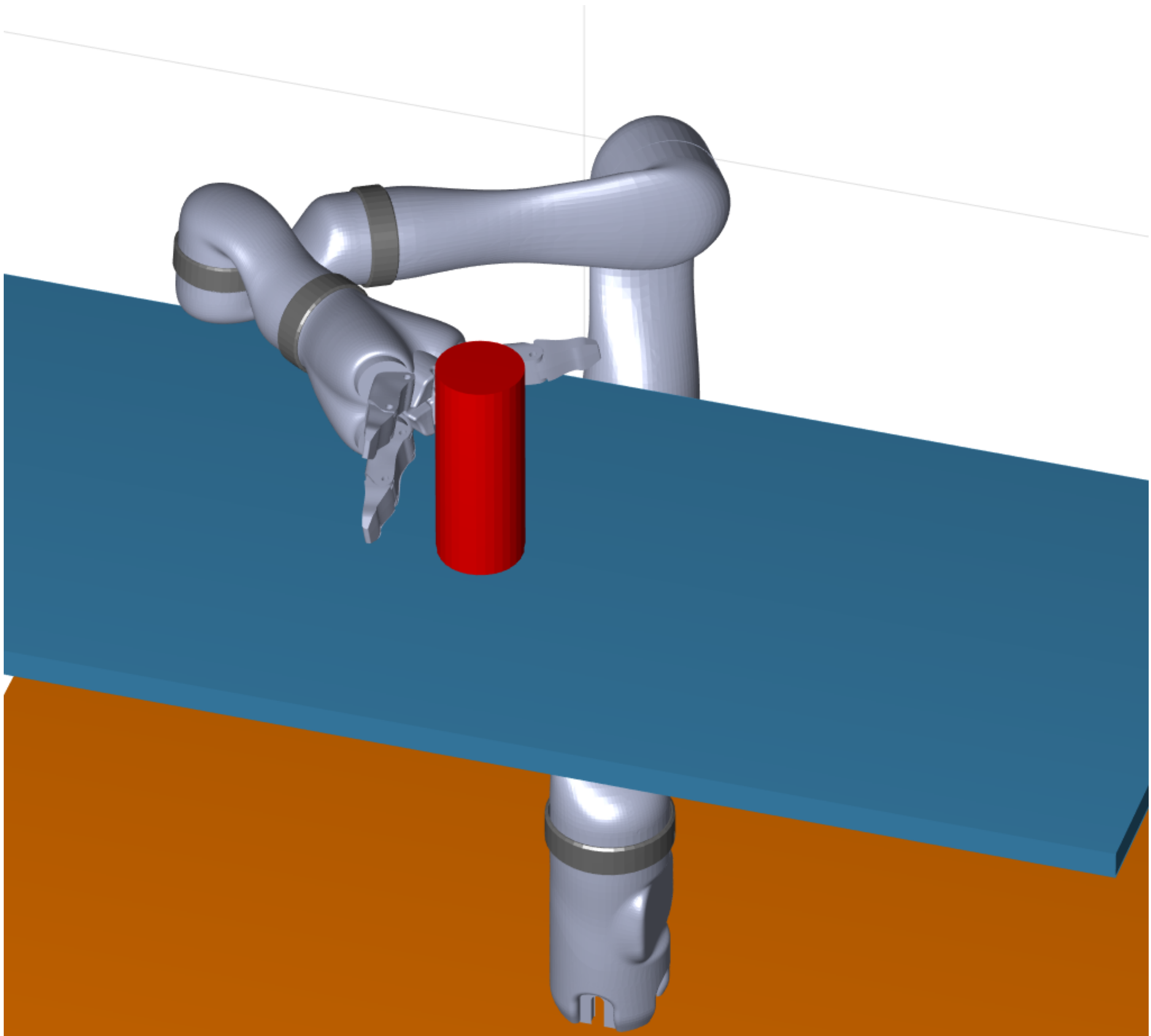
```
f = figure;

% 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 with RRT for a Robot Manipulator

This example shows how to plan a grasping motion for a **Kinova Jaco Assitive 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 valdiation for motion planning applications.



Define the manipulator

Load Kinova Jaco model from robot library. This particular model includes the three-finger gripper.

```
kin = loadrobot('kinovaJacoJ2S7S300');
```

Create the Environment

Using collision object primitives, add a floor, table top, and cylinder. Specify the size and pose of these objects. This code creates the scene shown in the image at the beginning of this example.

```
floor = collisionBox(1, 1, 0.01);
tabletop = collisionBox(0.4,1,0.02);
tabletop.Pose = trvec2tform([0.3,0,0.6]);
can = collisionCylinder(0.03,0.16);
can.Pose = trvec2tform([0.3,0.0,0.68]);
```

Customize the State Space for Manipulator

The Kinova arm has 10 degrees of freedom (DoFs), with the last three DoFs corresponding to the fingers. We only use the first 7 DoFs for the planning and keep the fingers at zero coniguration (i.e. open wide). An `ExampleHelperRigidBodyTreeStateSpace` state space is created to represent the `R7` configuration space (joint space). `ExampleHelperRigidBodyTreeStateSpace` is configured to sample 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 (i.e. work space) so that RRT can converge to a solution faster instead of getting lost in the 7 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
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,tabletop);
```

```
addFixedObstacle(sv,can);
addFixedObstacle(sv,floor);

% 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 configrations 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 conifiguration. Note this is applicable only when goal bia
Te_0ref = Tw_0*Te_w; % Reference end-effector pose in world coordinates, derived from W
ik = inverseKinematics('RigidBodyTree',kin);
refGoalConfig = ik(ss.EndEffector,Te_0ref,ones(1,6),homeConfiguration(ss.RigidBodyTree)

% Compute the initial conifiguration (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 = 0.3;
```

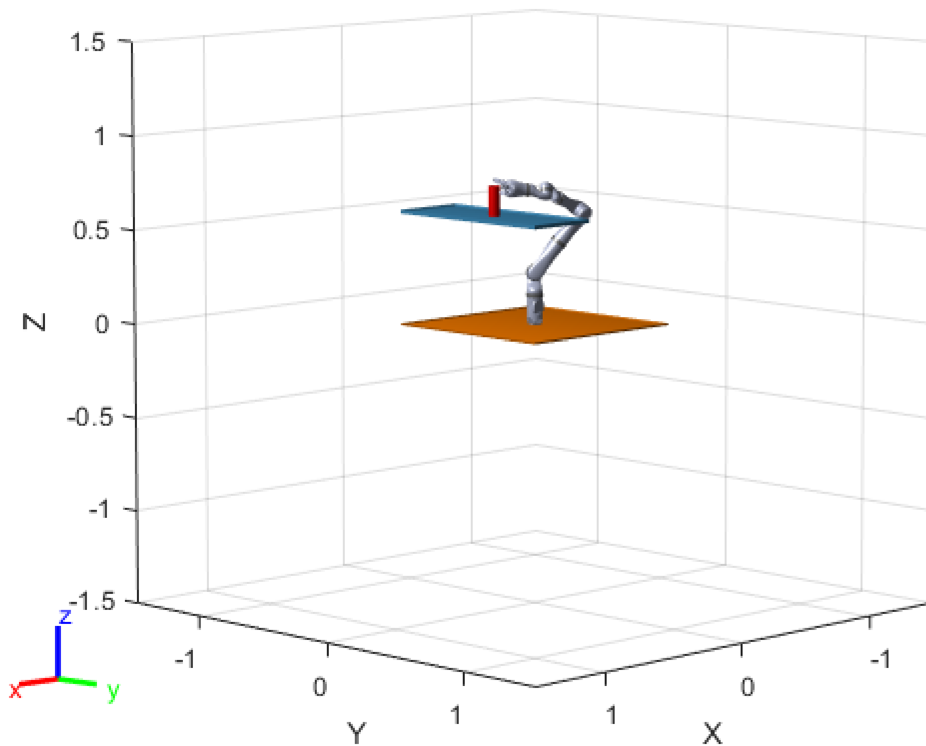
```
% turn off GoalBias for now
planner.GoalBias = 0;
[pthObj,solnInfo] = plan(planner,initConfig,refGoalConfig)

pthObj =
  navPath with properties:

    StateSpace: [1×1 ExampleHelperRigidBodyTreeStateSpace]
        States: [21×10 double]
     NumStates: 21


solnInfo = struct with fields:
      IsPathFound: 1
         ExitFlag: 1
         NumNodes: 42
    NumIterations: 186
         TreeData: [128×10 double]
```

Visualize the Grasping 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,:));

% Render the environment
hold on
[~,pFloor] = show(floor,'Parent',ax);
[~,pTable] = show(tabletop,'Parent',ax);
[~,pCan] = show(can,'Parent',ax);
pFloor.LineStyle = 'none';
pTable.LineStyle = 'none';
pCan.LineStyle = 'none';
pTable.FaceColor = [71 161 214]/256;
```

```
pCan.FaceColor = 'r';

% Show the motion
for i = 2:length(states)
    show(kin,states(i,:),'PreservePlot',false,'Frames','off','Parent',ax);
    drawnow
end
hold off
```



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.

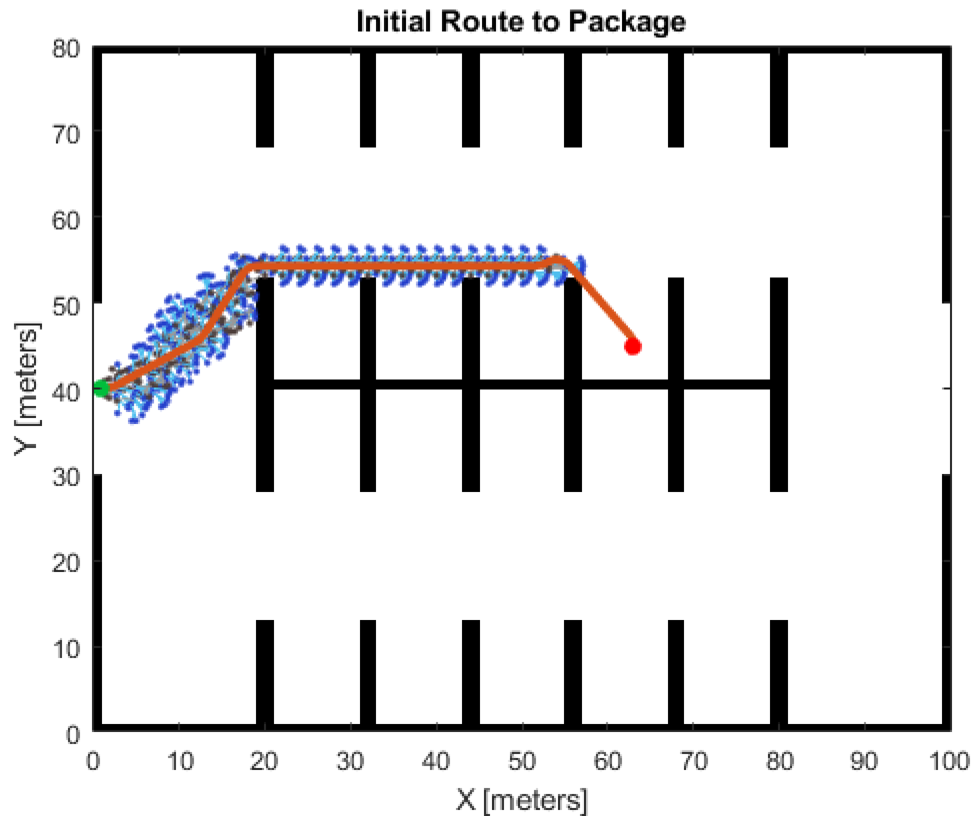
```
map = binaryOccupancyMap(100, 80, 1);
occ = zeros(80, 100);

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.

```
estMap = occupancyMap(occupancyMatrix(map));
vMap = validatorOccupancyMap;
vMap.Map = estMap;
planner = plannerHybridAStar(vMap, 'MinTurningRadius', 2);

entrance = [1 40 0];
packagePickupLocation = [63 45 -pi/2];
route = plan(planner, entrance, packagePickupLocation);
route = route.States;
```

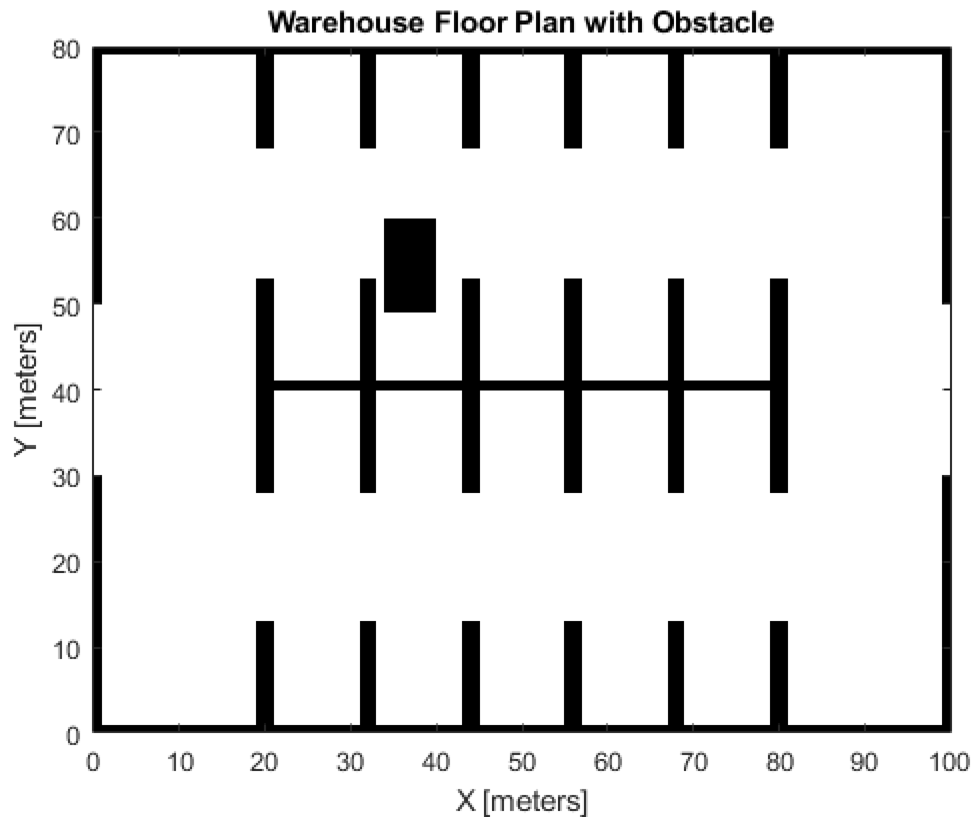
```
% Get poses from the route.
rsConn = reedsSheppConnection('MinTurningRadius', planner.MinTurningRadius);
startPoses = route(1:end-1,:);
endPoses = route(2:end,:);

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/2);
numReadings = rangefinder.NumReadings;
```

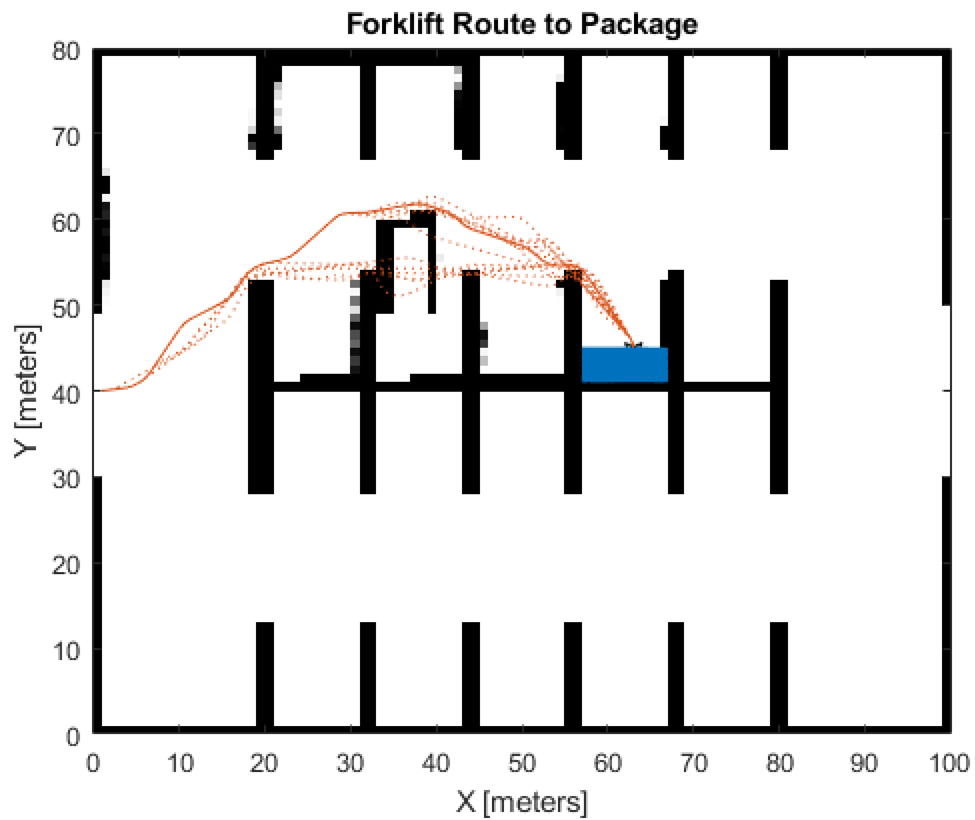
Update the Route Based on Range Finder Readings

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(estMap)
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 map.
    [ranges, angles] = rangefinder(poses(idx,:), map);
    insertRay(estMap, poses(idx,:), ranges, angles, ...
        rangefinder.Range(end));

    % Update visualization.
    show(estMap, '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(estMap, poses(:,1:2)));
    if isRouteOccupied && (toc > 0.5)
        % Calculate new route.
        planner.StateValidator.Map = estMap;
        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
```



Lane Change for Highway Driving

This example shows how to simulate an automated lane change maneuver system for highway driving scenario.

In this example, you will:

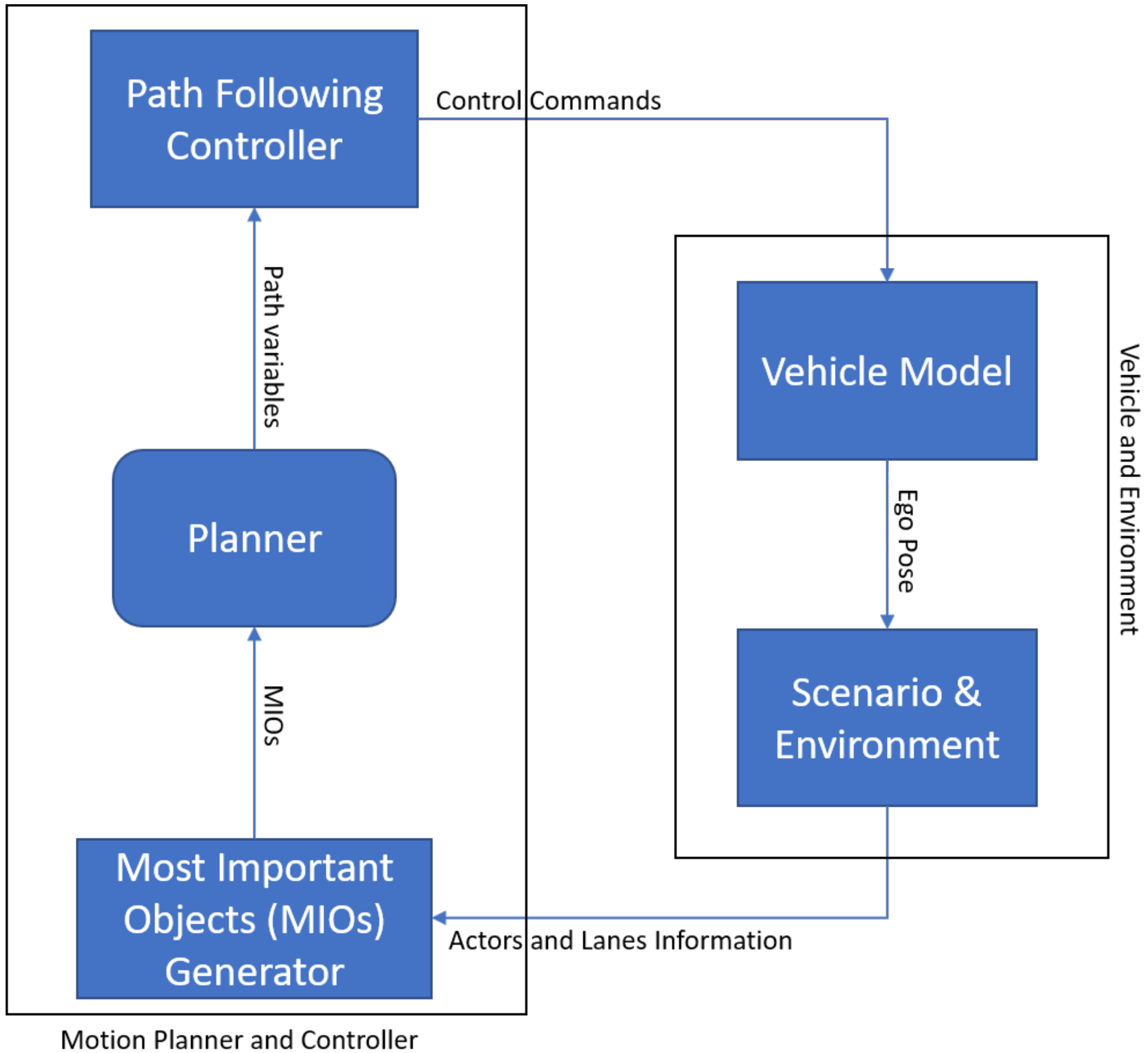
1. Perform trajectory planning to automate lane change maneuver.
2. Deploy a path following controller to steer the vehicle along the generated trajectory.
3. Test the lane change maneuver in a closed-loop Simulink® model using prebuilt scenarios generated by the Automated Driving Toolbox™.

Contents

- Introduction on page 1-0
- Open Test Bench Model on page 1-0
- Explore Planner and Controller Subsystem on page 1-0
- Explore Vehicle and Environment Subsystem on page 1-0
- Explore Metric Assessment Subsystem on page 1-0
- Simulate and Visualize System Behavior on page 1-0
- Conclusion on page 1-0

Introduction

An automated lane change maneuver (LCM) system assists the ego vehicle to automatically move from one lane to another lane (with the same direction of travel) in a highway. The LCM system models the longitudinal and lateral control dynamics for automated lane change on a straight road segment. The high-level block diagram of the LCM system is as shown in the figure below



The LCM system dynamically generates an optimal trajectory for lane change while avoiding collision with other actors in the scenario. The LCM system steers the ego vehicle to travel along the optimal trajectory. The LCM system test bench model focuses

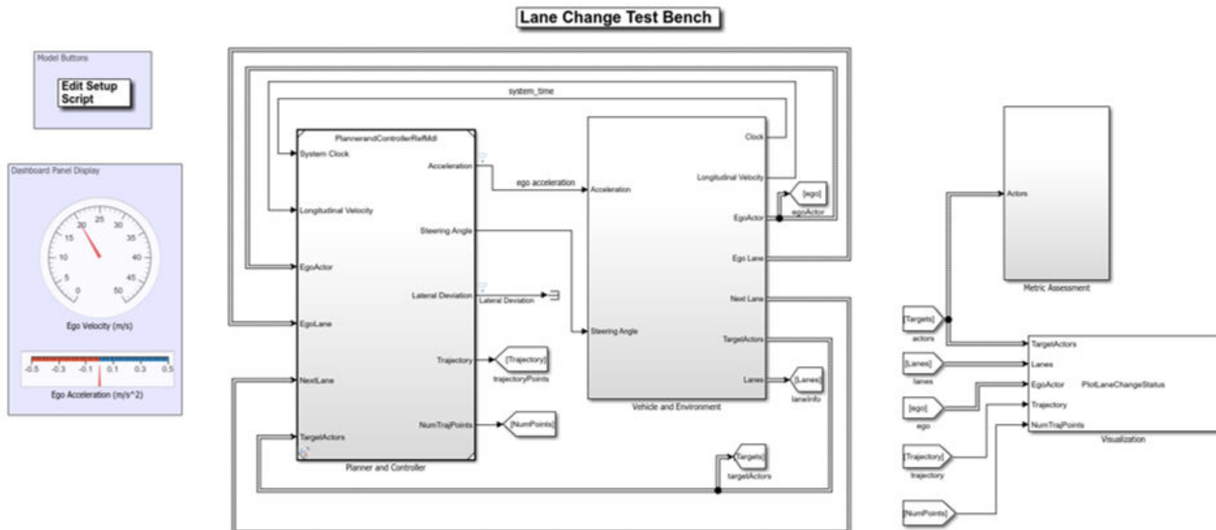
on the planner and the controller aspects of a lane change system. The model comprises of three main subsystems

1. **Planner and Controller** subsystem generates the optimal trajectory for lane change and sends a control command to steer the ego vehicle.
2. **Vehicle and Environment** subsystem models the environment and the motion of the ego vehicle. The subsystem receives control command from the **Planner and Controller** subsystem and updates the state for the ego vehicle.
3. **Metric Assessment** subsystem validates the state of the ego vehicle for collision avoidance.

Open Test Bench Model

To open the Simulink test bench model, use the following command.

```
open_system('LaneChangeTestBench')
```

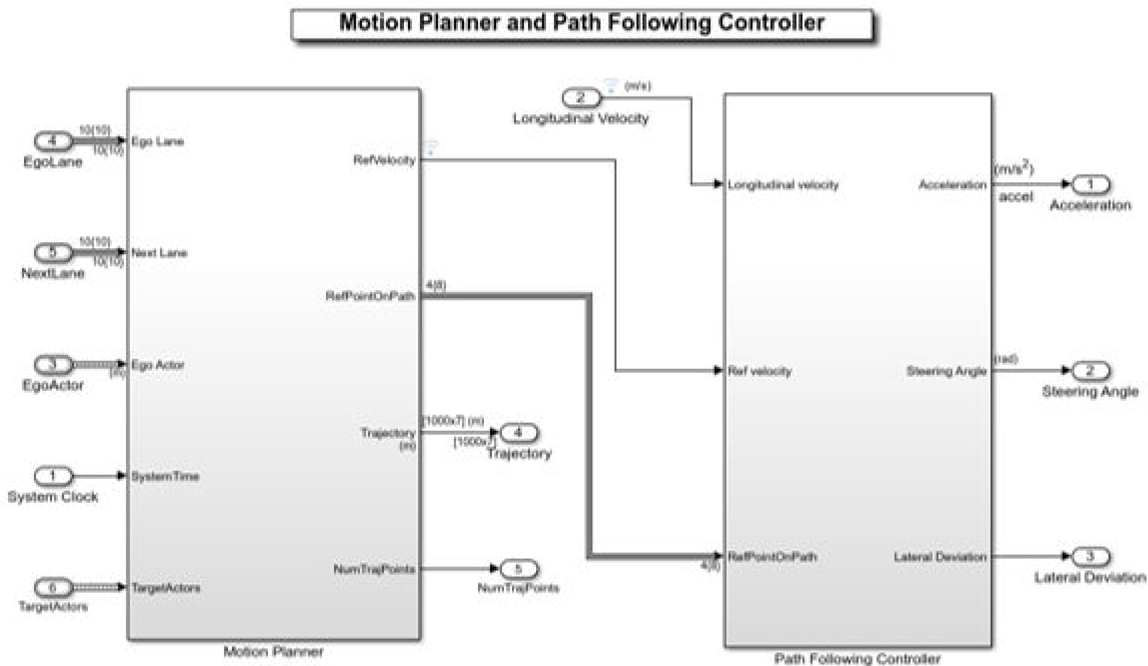


Opening this model runs the `exampleHelperLaneChangeSetup` script, which initializes the vehicle model parameters, controller design parameters, and the road scenario. The script also initializes the buses required for defining the inputs into and outputs for the LCM Simulink model.

Explore Planner and Controller Subsystem

Open the planner and controller model under test.

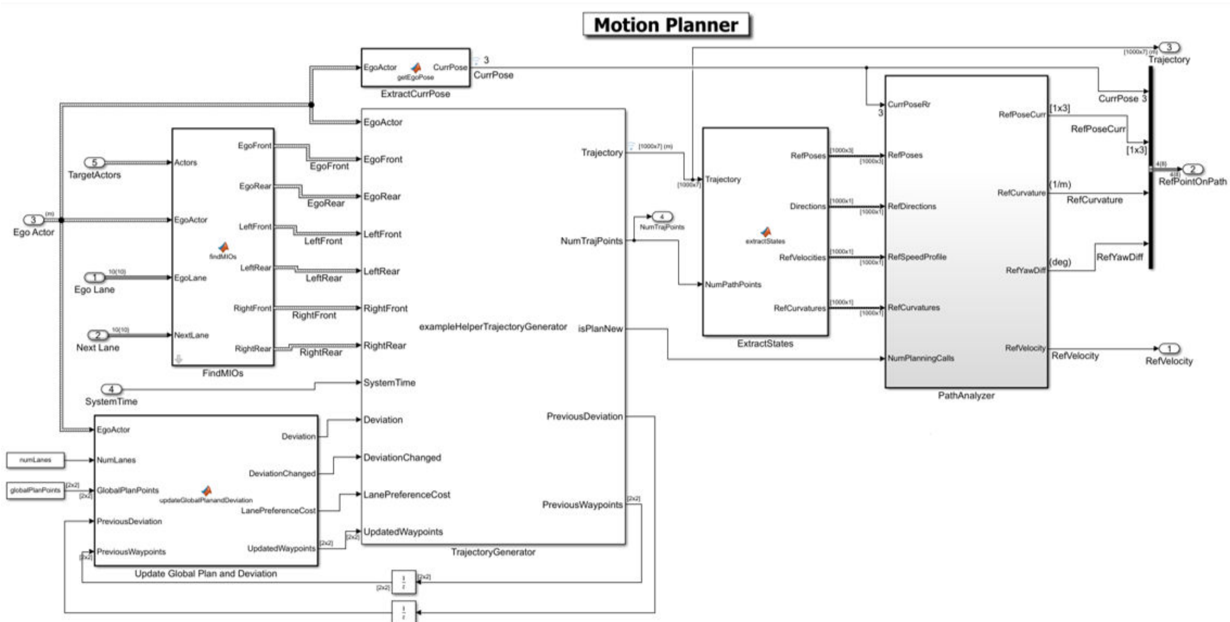
```
open_system('PlannerandControllerRefMdl')
```



The planner and the controller subsystem comprise of a **Motion Planner** and a **Path Following Controller** that simulates trajectory planning and motion control of the ego vehicle respectively.

A) Motion Planner

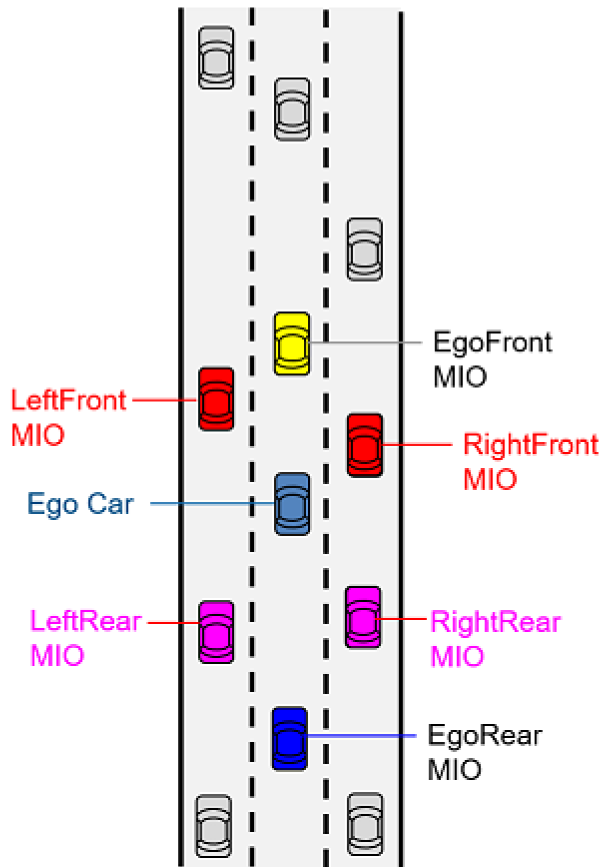
```
open_system('PlannerandControllerRefMdl/Motion Planner');
```



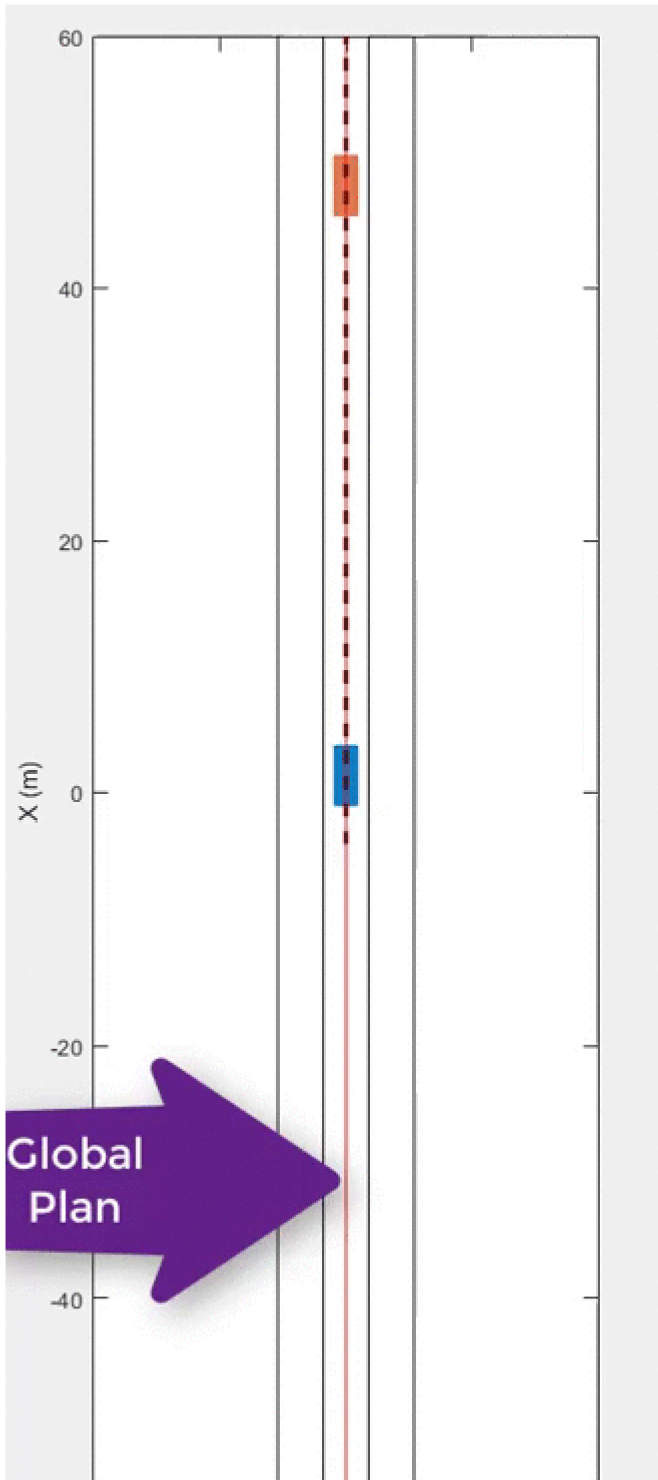
The **Motion Planner** performs trajectory planning by using the lane information, ego data, and the non-ego actors' data from the Vehicle and Environment subsystem block. The **Motion Planner** consists of:

1. **Find MIOs** function block
2. **Update Global Plan and Deviation** function block
3. **Trajectory Generator** subsystem
4. **Trajectory to Path** function block
5. **Path Analyzer** subsystem

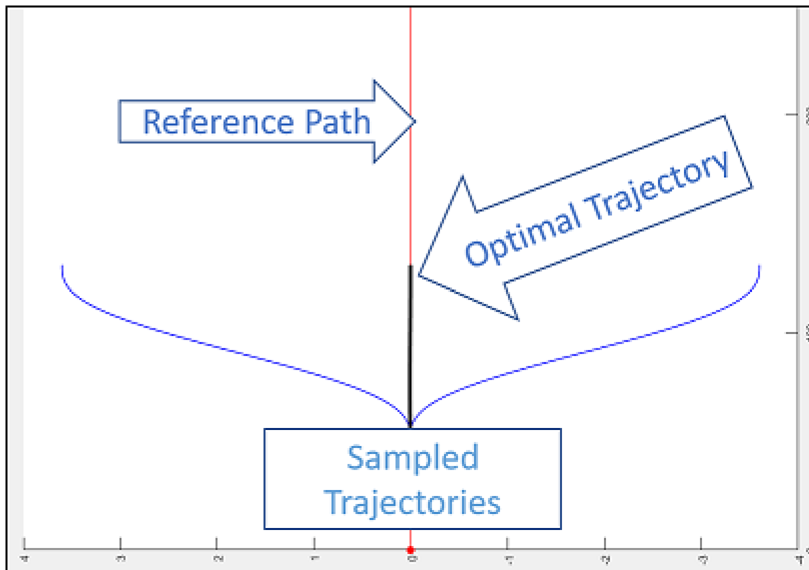
- The **Find MIOs** MATLAB function block finds the most important objects (MIOs) in the scenario. MIOs refer to the non-ego actors that lie in the vicinity of the ego vehicle. The MIOs can be located in front or rear of the ego-vehicle along its current lane and the adjacent lanes.



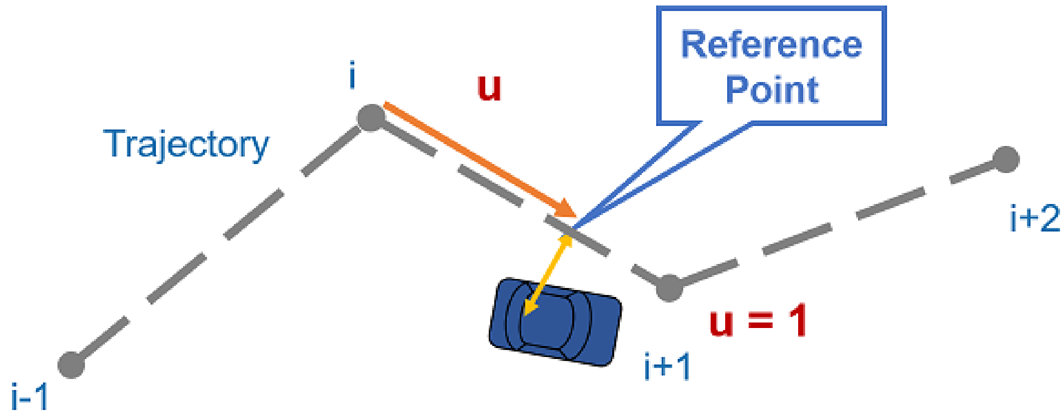
- The **Update Global Plan and Deviation** MATLAB function block calculates the desired deviation from the reference path and updates the global plan to prevent frequent lane changes and maintain the lateral deviation. The global plan updates based on the deviation. A positive offset value means the ego vehicle is offset to the right of the reference path. A negative offset value means the ego vehicle is offset to the left of the reference path.


60
40
20
0
-20
-40
X (m)
Global
Plan

- The **Trajectory Generator** subsystem generates an optimal trajectory based on MIOs, reference path, and the specified deviation. MIOs are used to update the state validator. The state validator object and the reference path waypoints are passed as inputs to the `trajectoryOptimalFrenet` function. The `trajectoryOptimalFrenet` function generates multiple trajectories. Sampled trajectories are validated for dynamic collision using the state validator. The subsystem identifies the optimal trajectory based on different weights and set properties of `trajectoryOptimalFrenet`.



- The **Trajectory to Path** MATLAB function block extracts path information from the generated trajectory.
- The **Path Analyzer** subsystem estimates the heading angle and finds the appropriate point on the path to follow. The generated path must conform to the road shape.



B) Path Following Controller

The **Path Following Controller** simulates a path-following control (PFC) mechanism that keeps the ego vehicle traveling along the generated trajectory while tracking a set velocity. To do so, the controller adjusts both the longitudinal acceleration and front steering angle of the ego vehicle. The controller computes optimal control actions while satisfying velocity, acceleration, and steering angle constraints using adaptive model predictive control (MPC). The controller comprises of:

- A **Virtual Lane Center** subsystem - creates a virtual lane from the path point. The virtual lane matches the format required by the path following controller.
- A **Preview Curvature** subsystem - converts trajectory to curvature input required by path following controller.
- A **Lane Following Controller** subsystem - uses the Path Following Control System block from the Model Predictive Control Toolbox.

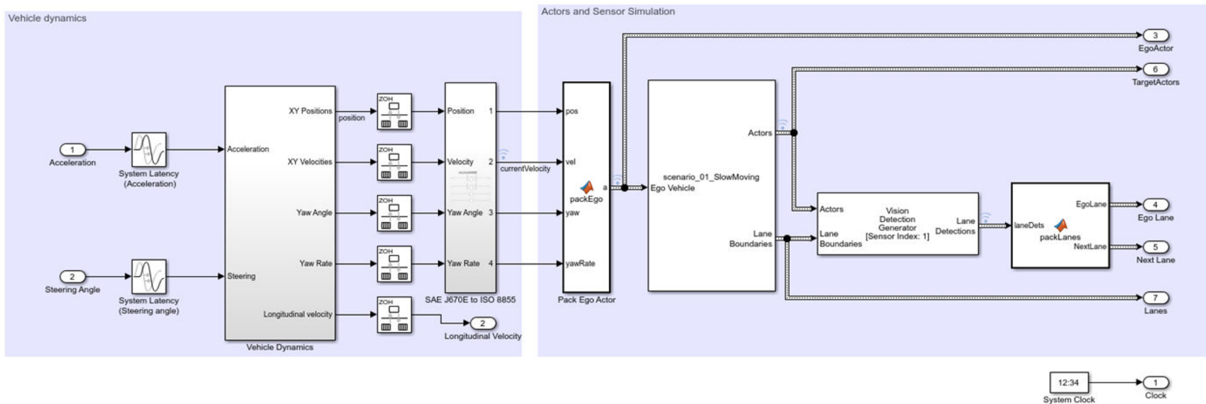
The **Lane Following Controller** keeps the vehicle traveling within a marked lane of a highway while maintaining a user-set velocity or safe distance from the preceding vehicle. The lane following controller includes combined longitudinal and lateral control of the ego vehicle:

- Longitudinal control - Maintain a user-set velocity and keep a safe distance from the preceding car in the lane by adjusting the acceleration of the ego vehicle.
- Lateral control - Keep the ego vehicle travelling along the centerline of its lane by adjusting the steering of the ego vehicle.

Explore Vehicle and Environment Subsystem

The **Vehicle and Environment** subsystem enables closed-loop simulation of the path following controller.

```
open_system('LaneChangeTestBench/Vehicle and Environment')
```

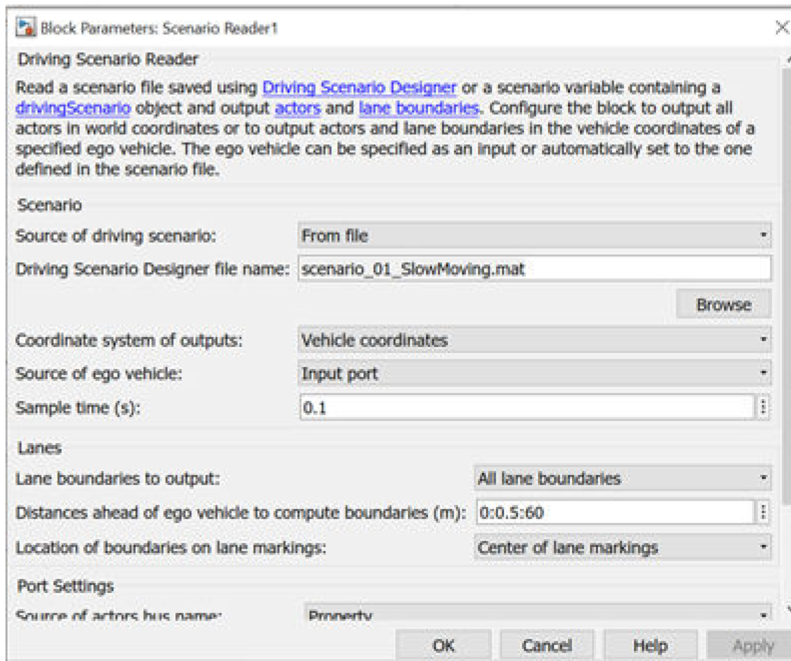


The **System Latency** blocks model the latency in the system between model inputs and outputs.

The **Vehicle Dynamics** subsystem models the vehicle dynamics using a Bicycle Model - Force Input block from the Vehicle Dynamics Blockset™. The lower-level dynamics are modeled by a first-order linear system with a time constant of seconds.

The **SAE J670E to ISO 8855** subsystem converts the coordinates from Vehicle Dynamics, which uses SAE J670E, to Scenario Reader, which uses ISO 8855.

The **Scenario Reader** block reads data from a scenario file created using the Driving Scenario Designer app. To change the scenario file, double-click on the **Scenario Reader** block and open the block parameters tab.

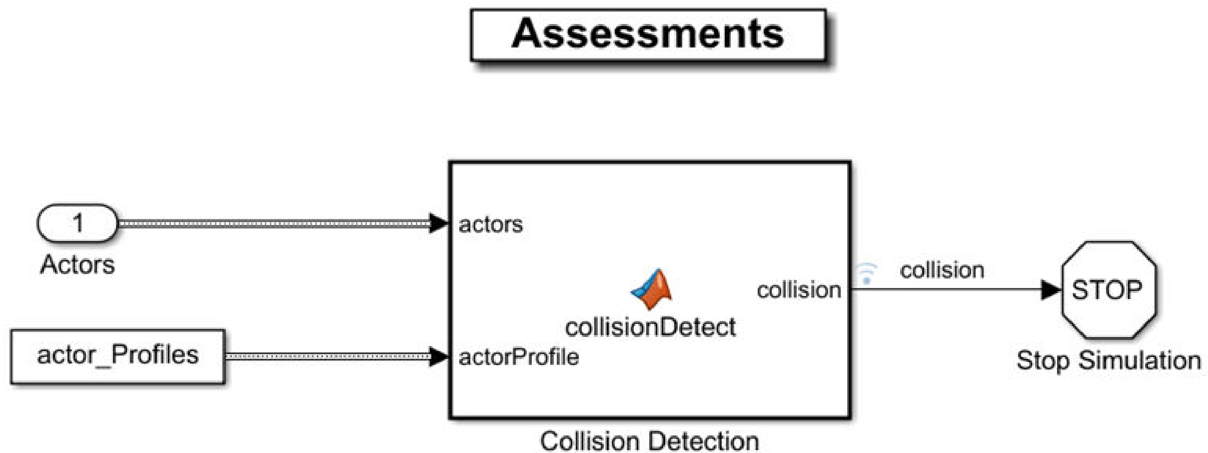


Set **Source of driving scenario** parameter to `From File` and click on **Browse** to select the desired scenario file. The **Scenario Reader** block reads the actor data and the lane information from the scenario file. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates.

The **Vision Detection Generator** block takes lane boundaries from the **Scenario Reader** block and generates detections from simulated actor poses. All detections are referenced to the coordinate system of the ego vehicle. The detection generator models the field of view of a monocular camera and determines the heading angle, curvature, curvature derivative, and valid length of each lane boundary, accounting for any other obstacles.

Explore Metric Assessment Subsystem

```
open_system('LaneChangeTestBench/Metric Assessment')
```

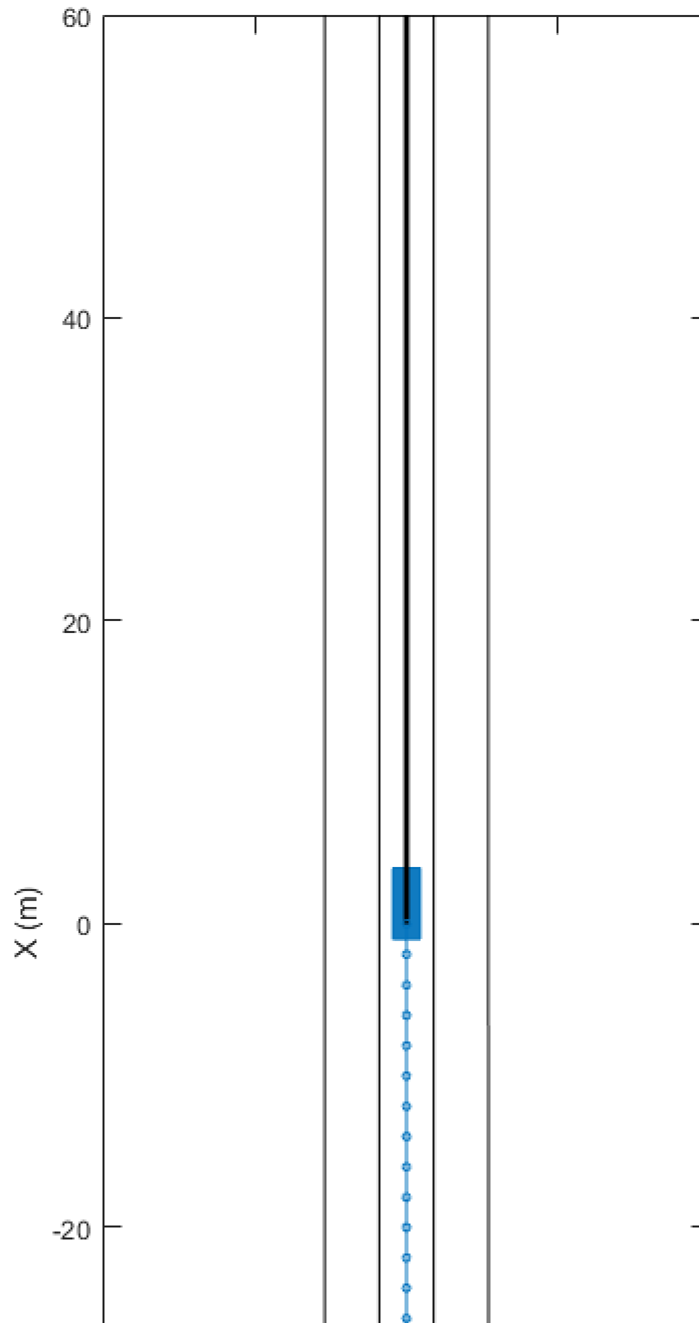


The **Metric Assessment** subsystem validates that the ego vehicle does not collide with other vehicles. The **Metric Assessment** subsystem halts lane change maneuvering and stops the simulation if the system detects a collision of ego vehicle with target vehicles.

Simulate and Visualize System Behavior

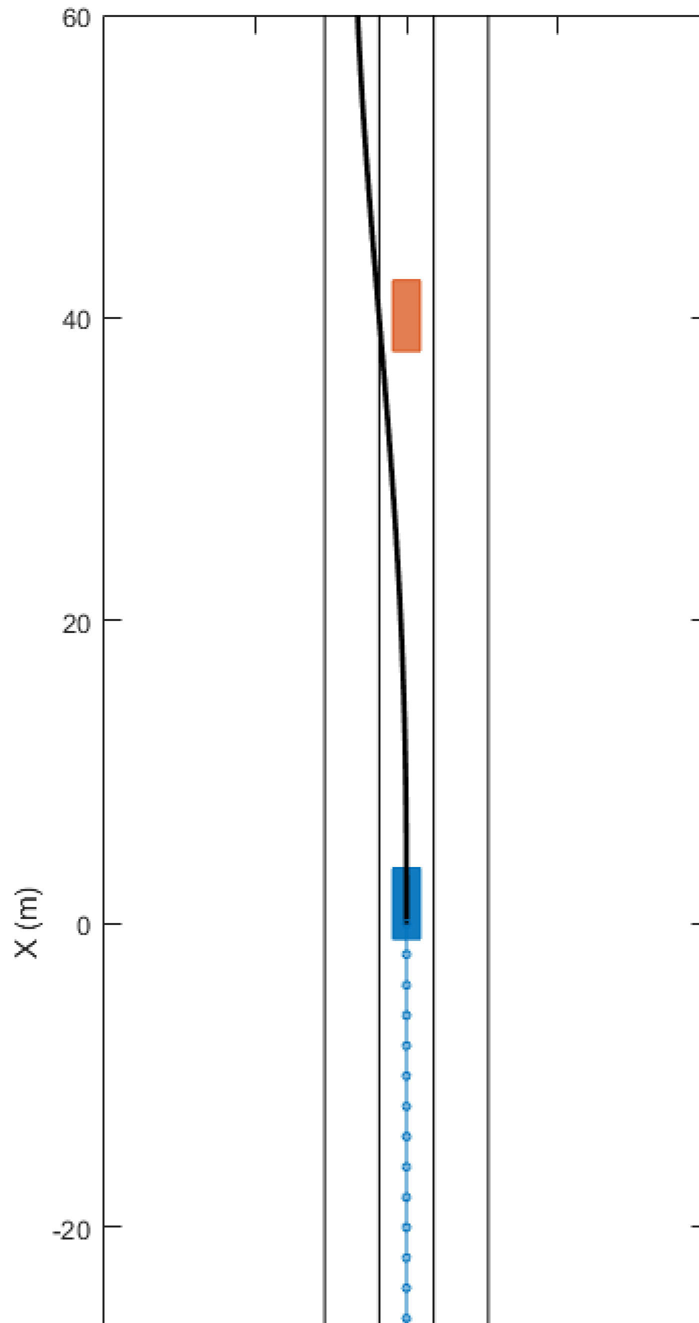
Run the LCM simulation model system to visualize the behavior of the system during a lane change. The **Visualization** block in the model creates a bird's eye plot that displays the lane information, ego vehicle, ego trajectory, and other vehicles in the scenario.

```
% Turn off messages from MPC.
mpcverbosity('off');
% Run the model for 3 secs.
sim("LaneChangeTestBench","StopTime","3");
```


60
40
20
0
-20
X (m)

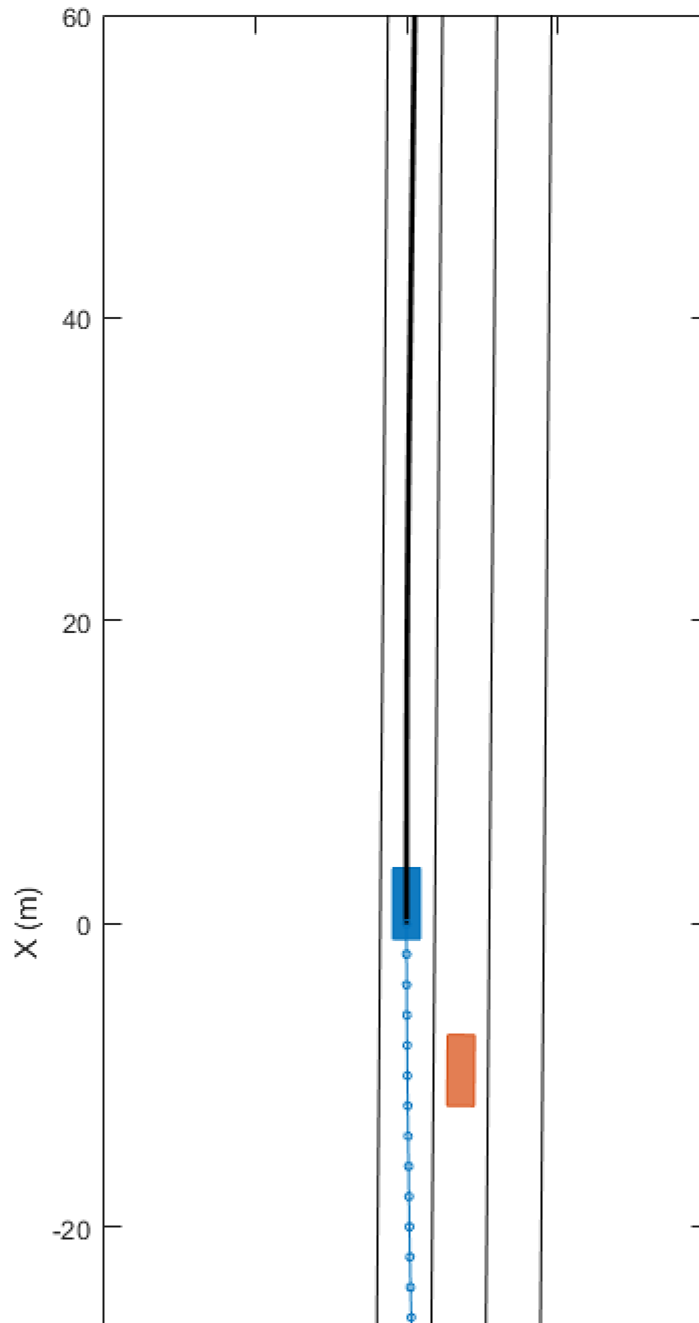
Run the simulation for 9 seconds and notice that a trajectory is calculated to navigate around a slower lead vehicle.

```
% Turn off messages from MPC.
mpcverbosity('off');
% Run the model for 3 secs.
sim("LaneChangeTestBench","StopTime","9");
```


60
40
20
0
-20
X (m)

Run the simulation for 14 seconds and notice that the vehicle continues straight ahead in the left lane.

```
% Turn off messages from MPC.
mpcverbosity('off');
% Run the model for 3 secs.
sim("LaneChangeTestBench","StopTime","14");
```


60
40
20
0
-20
X (m)

Explore the other test scenarios in the example by changing the `scenarioId` and running the `exampleHelperLaneChangeSetup` script. This script updates the scenario reader block with the selected scenario and reinitializes the buses with respect to the selected scenario. Save and run the model to explore the behavior. Updating the scenario from scenario reader block needs updating the `scenarioId` from the command line and re-run the `exampleHelperLaneChangeSetup` script to reinitialize buses required by the model.

Conclusion

This example showed how to simulate a highway lane change maneuver using ideal vehicle positions and lane detections.

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)

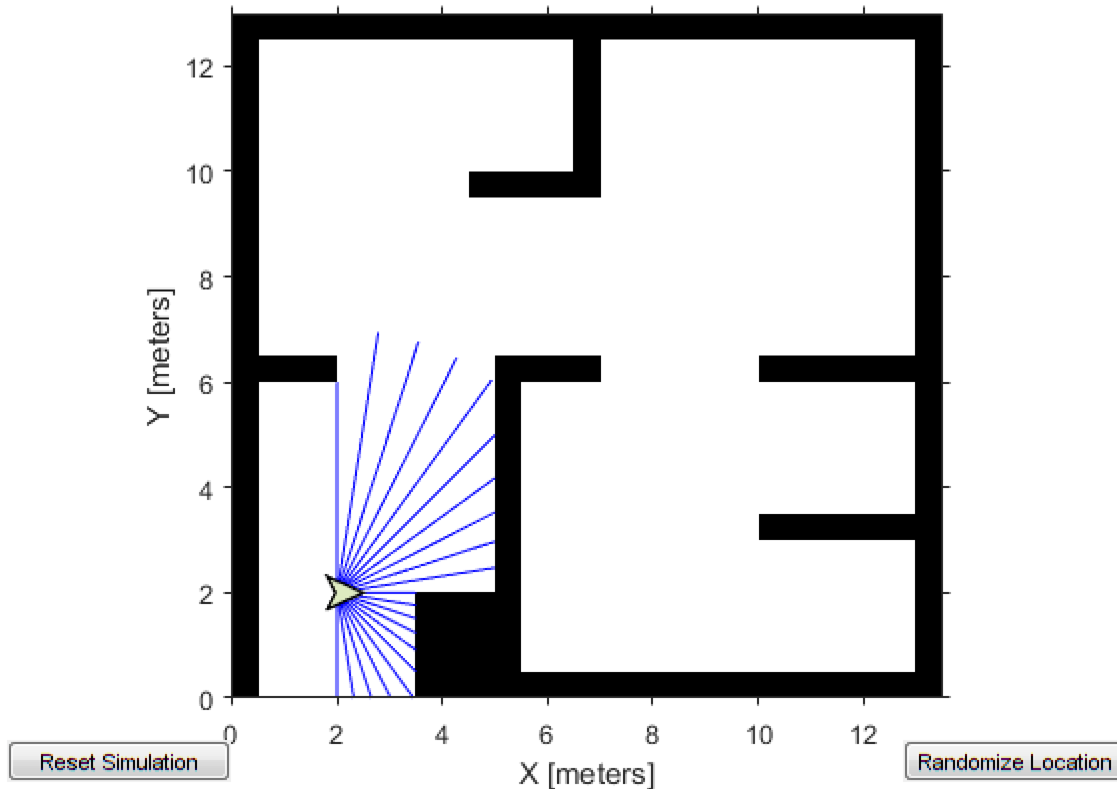
Introduction

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 `rosinit` 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 a 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. You also need a localization algorithm to get robot position in the Gazebo. See “Localize TurtleBot Using Monte Carlo Localization” on page 1-136 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
```

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 Obsta
```

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`.

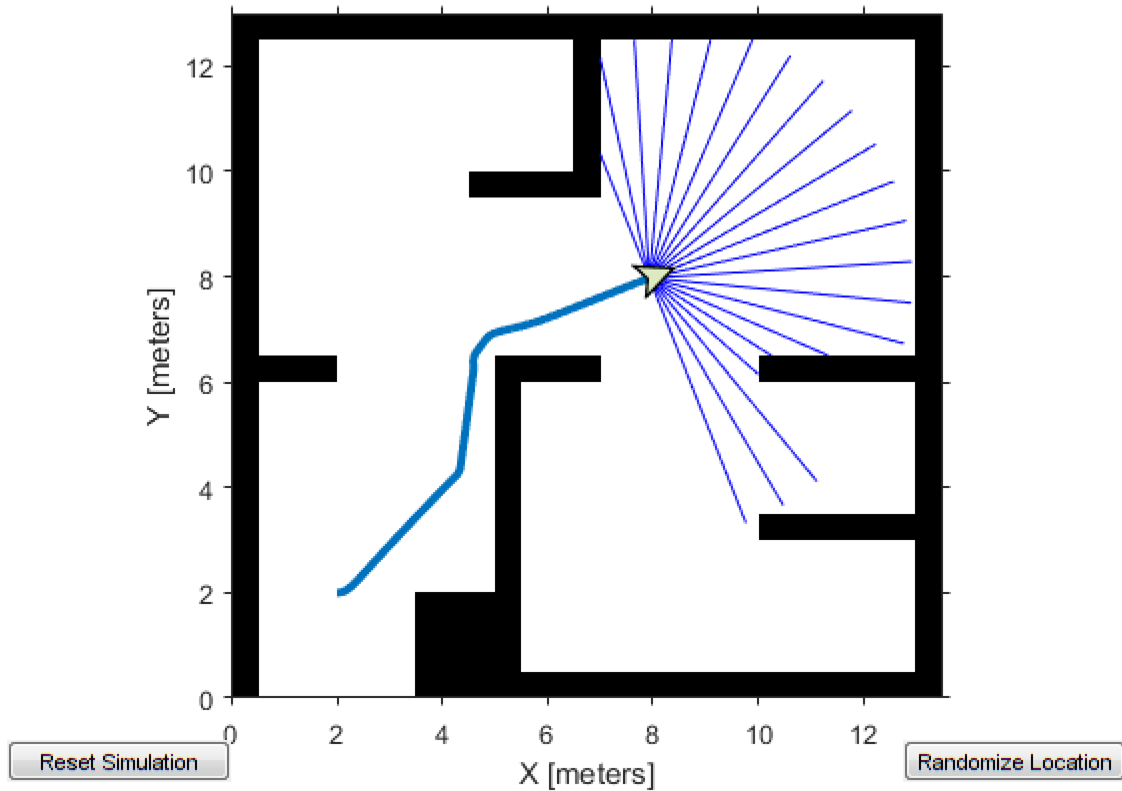
This is an enabled subsystem which is triggered when new laser message is received. This means a velocity command is published only when a new sensor information is available. This prevents the robot from hitting the obstacles in case of delay in receiving sensor information.

```
open_system('pathFollowingWithObstacleAvoidanceExample/Outputs','tab');
```

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 a 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://192.168.233.1:6
```

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

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_90736 with NodeURI http://192.168.233.1:6
```

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:

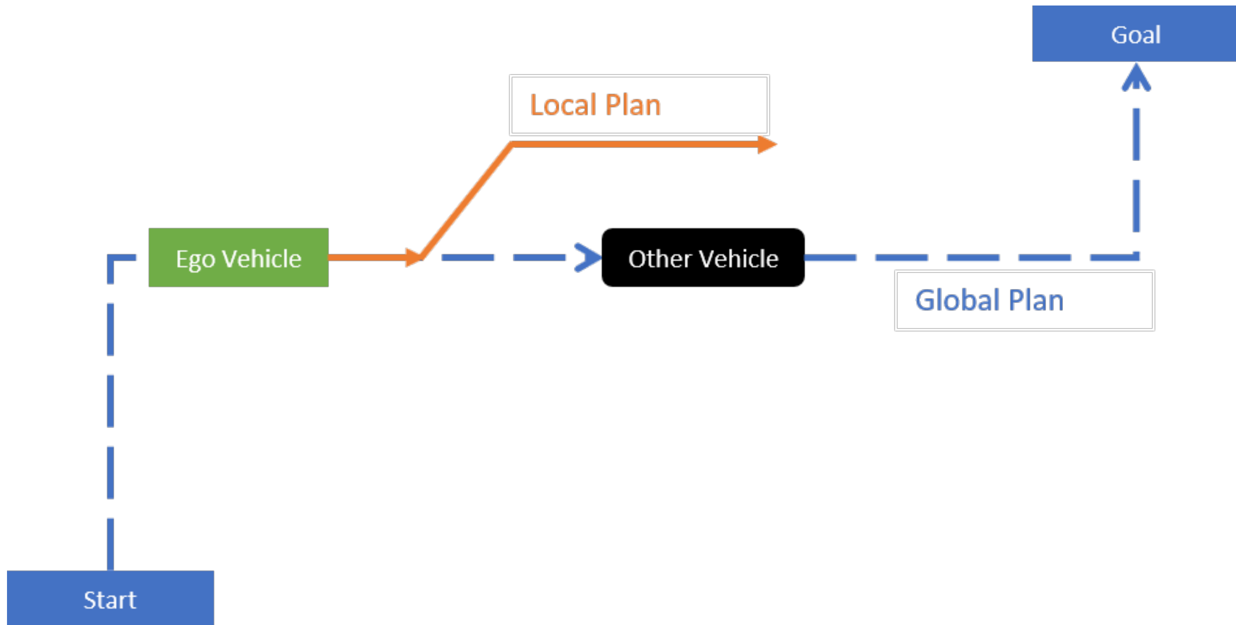
1. Explore an urban scenario with predefined vehicles.
2. Use `trajectoryOptimalFrenet` to do local planning for navigating the Urban scenario.

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

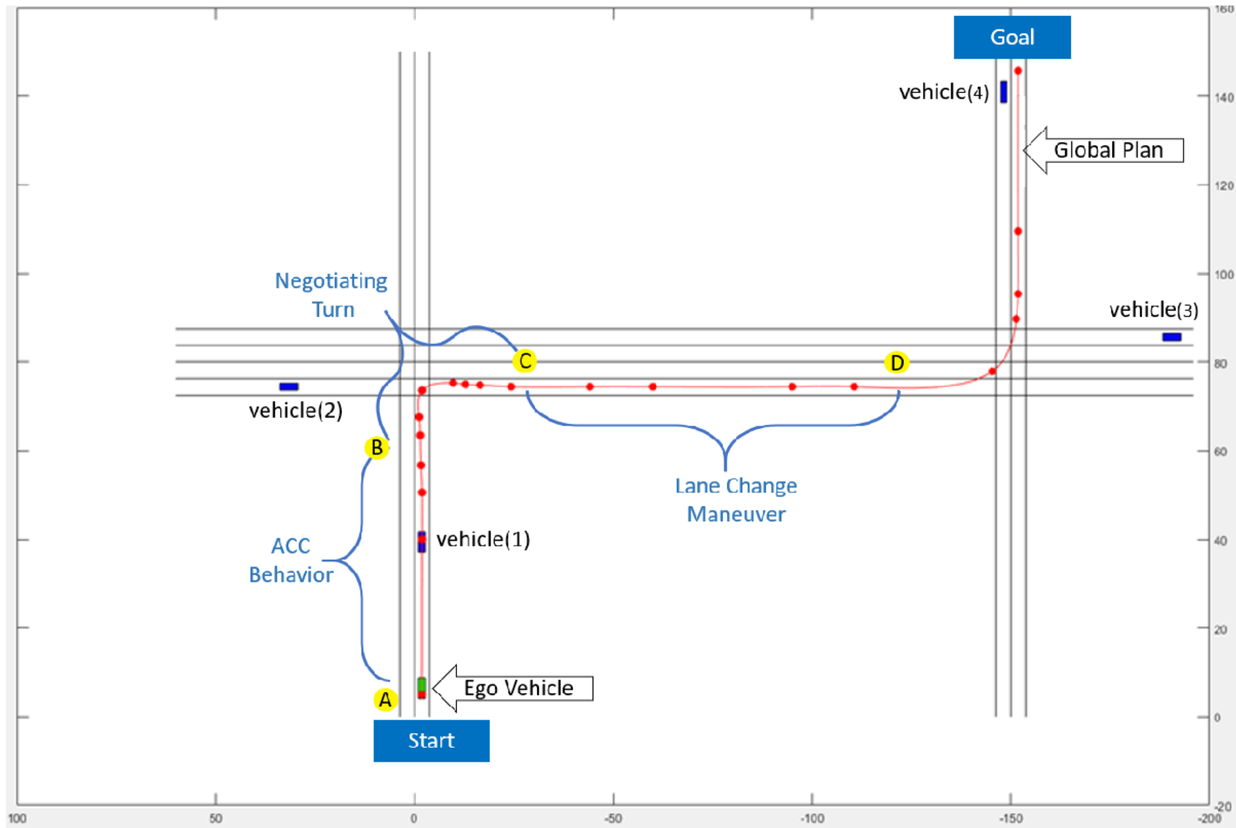
Automated driving in an urban scenario needs planning on two levels, global and local. The *global planner* finds the most feasible path between start and goal points. The *local planner* does dynamic replanning to generate an optimal trajectory based on the global plan and the surrounding information. 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:

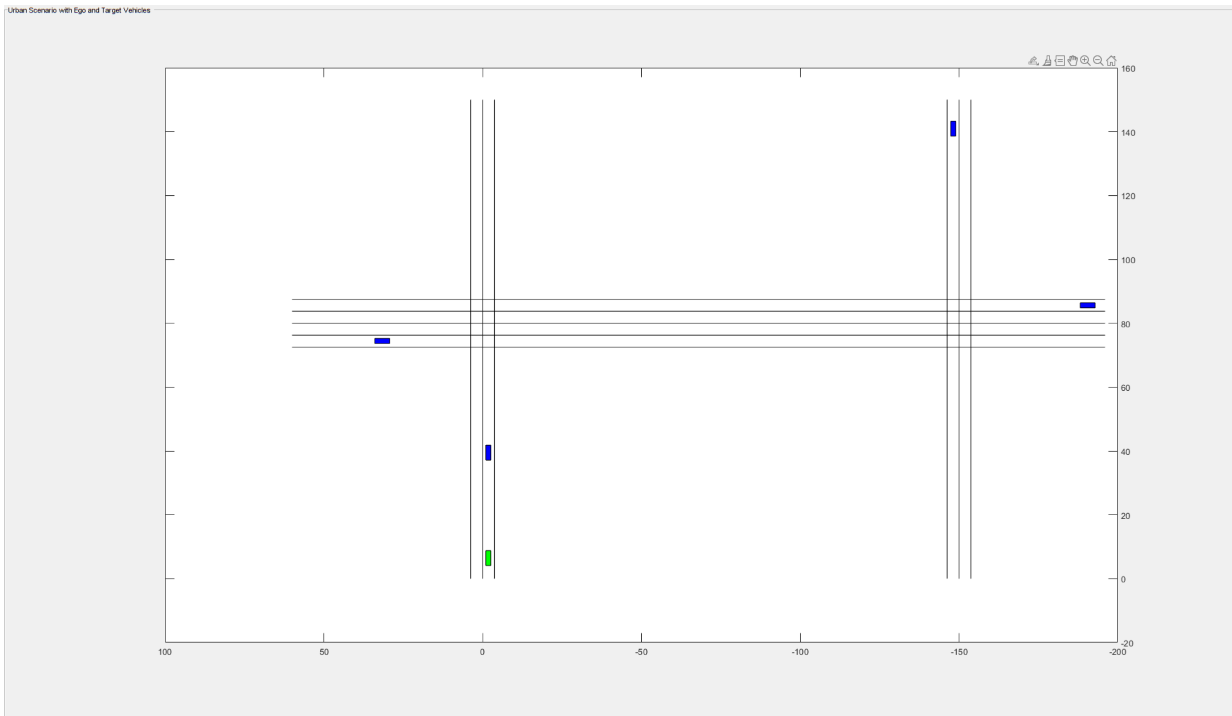
```
% Load data from urbanScenarioData.mat file, initialize required variables
[otherVehicles,globalPlanPoints,stateValidator] = exampleHelperUrbanSetup;
```

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 Postion 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;

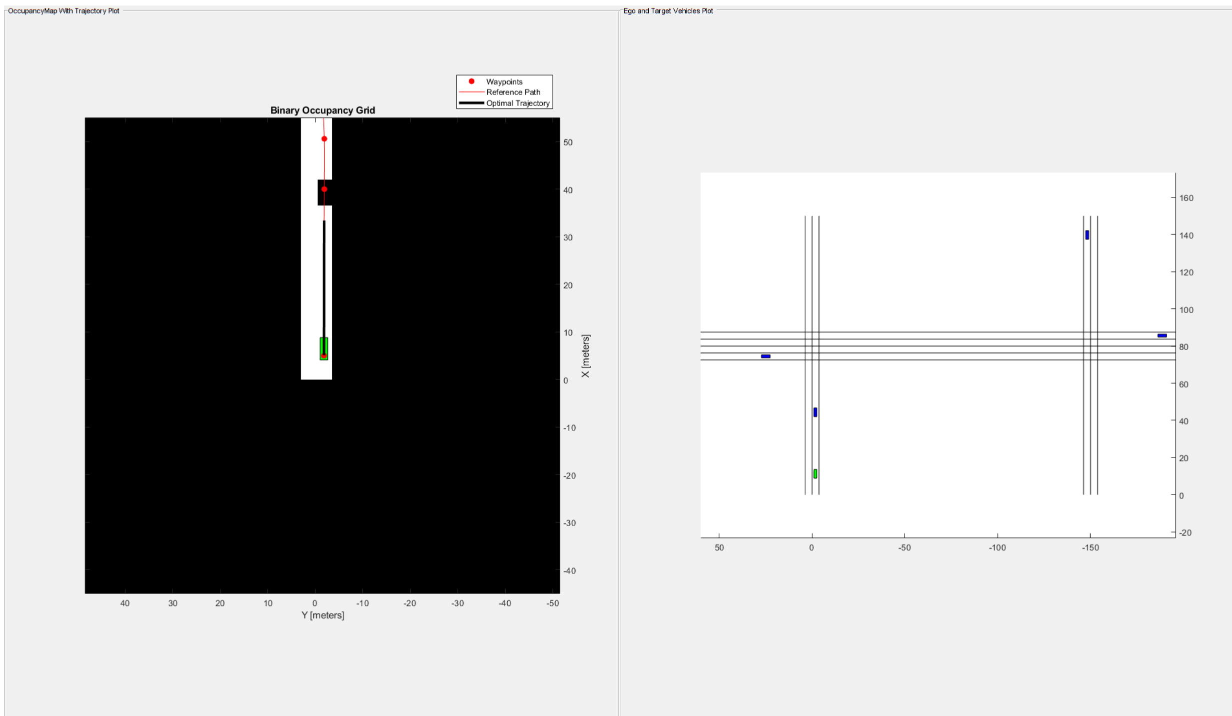
In the following code snippet, `localPlanner` generates `trajectory` that is executed and visualized using `exampleHelperUpdateVisualization` for every simulation step. However, replanning is done for every 100th simulation step. The following is the sequence of events during replanning:

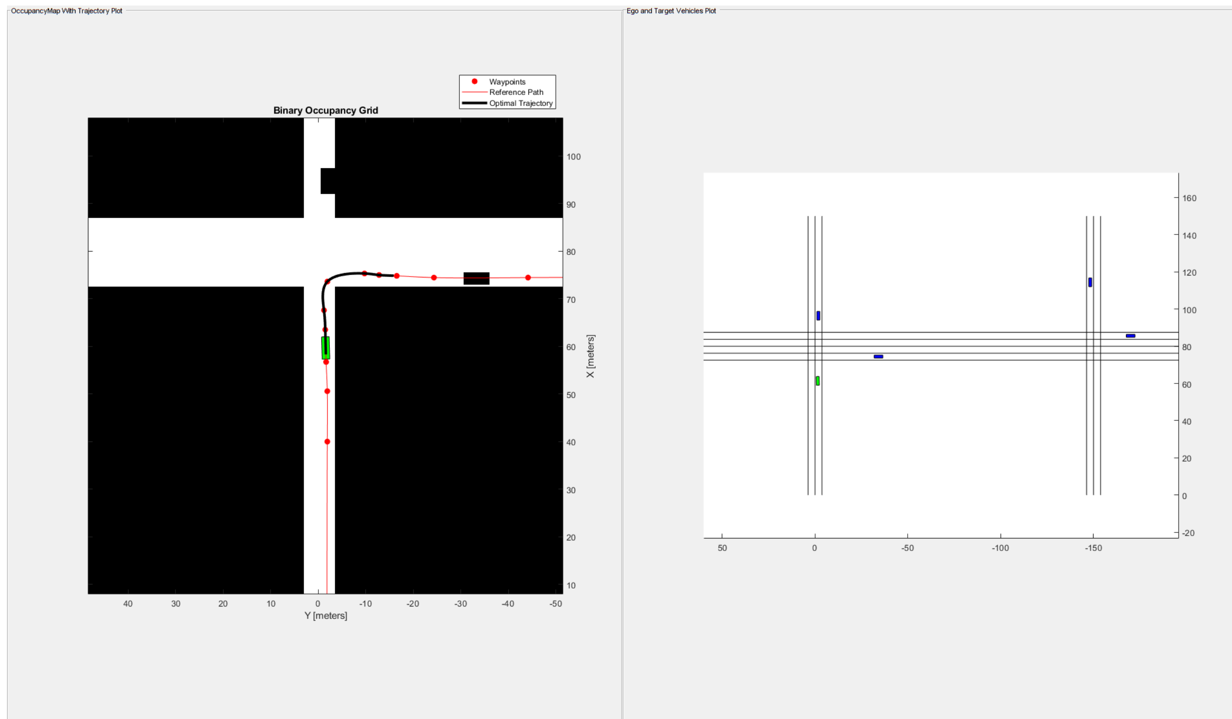
- Update the occupancy map using vehicle information using `exampleHelperUpdateOccupancyMap`.

- 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 - vehicleL
            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;
        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)
        title("Ego Centric Occupancy Map","Parent",hAxes1)

        % Visualize ego vehicle on occupancy map
        egoCenter = currentEgoState(1:2);
        egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)),
        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.

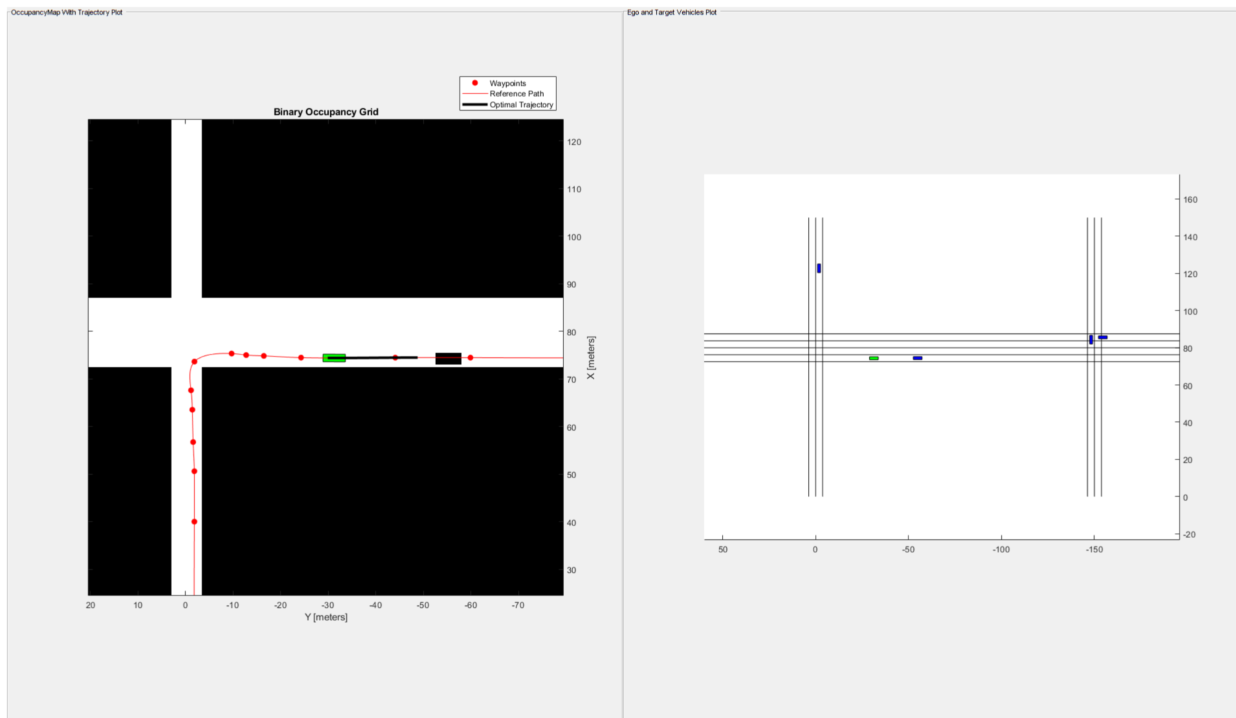
Use `trajectoryOptimalFrenet` to Negotiate a Smooth Turn

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 - vehicleL
        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)
    title("Ego Centric Occupancy Map","Parent",hAxes1)

    % Visualize ego vehicle on occupancy map
    egoCenter = currentEgoState(1:2);
    egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)),
    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.

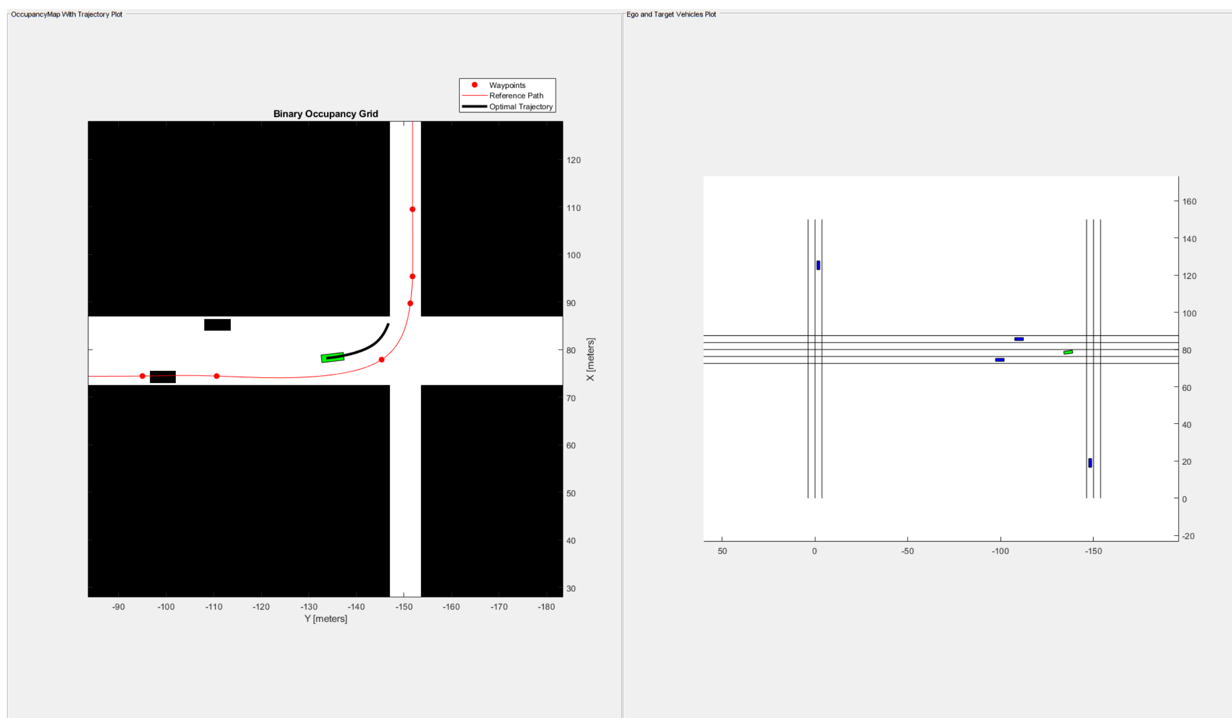
Next, configure `trajectoryOptimalFrenet` for performing a lane change maneuver from position C to position D.

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)
        title("Ego Centric Occupancy Map","Parent",hAxes1)

        % Visualize ego vehicle on occupancy map
        egoCenter = currentEgoState(1:2);
        egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)),
        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
```

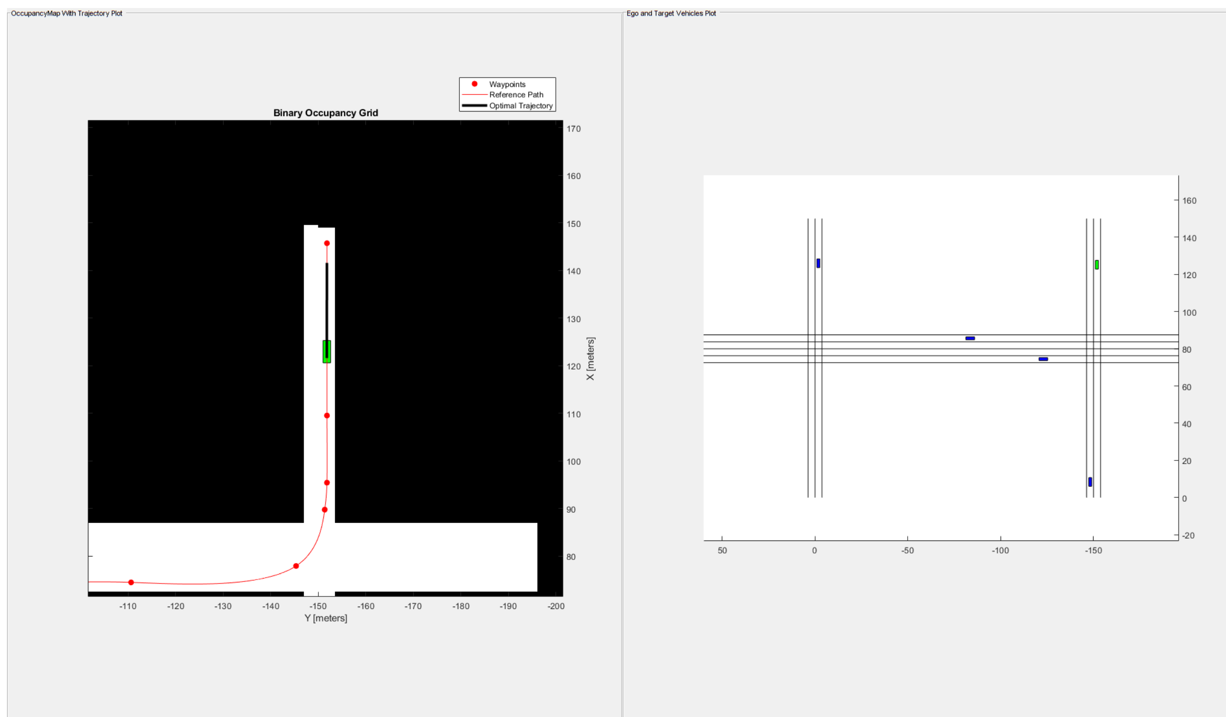


Simulate ego vehicle execution to reach the goal point

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)
        title("Ego Centric Occupancy Map","Parent",hAxes1)

        % Visualize ego vehicle on occupancy map
        egoCenter = currentEgoState(1:2);
        egoPolygon = exampleHelperTransformPointtoPolygon(rad2deg(currentEgoState(3)),
        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.

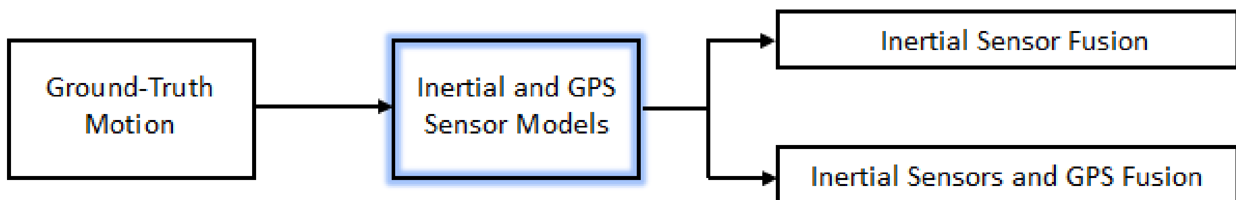
Navigation Topics

- “Model IMU, GPS, and INS/GPS” on page 2-2
- “Occupancy Grids” on page 2-10
- “Execute Code at a Fixed-Rate” on page 2-21
- “Particle Filter Workflow” on page 2-24
- “Particle Filter Parameters” on page 2-29
- “Pure Pursuit Controller” on page 2-36
- “Monte Carlo Localization Algorithm” on page 2-39
- “Vector Field Histogram” on page 2-50

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.

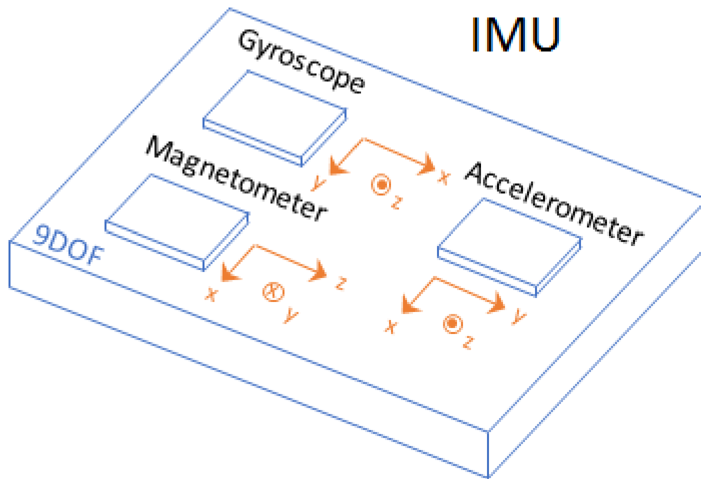
This tutorial provides an overview of inertial sensor and GPS models in Navigation Toolbox.



To learn how to generate the ground-truth motion that drives the sensor models, see `waypointTrajectory` and `kinematicTrajectory`.

Inertial Measurement Unit

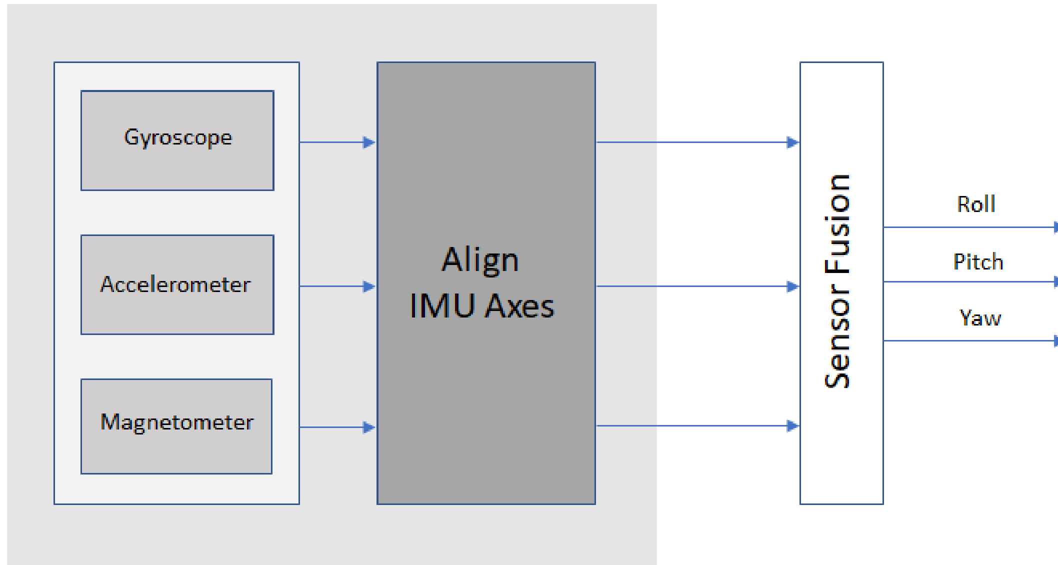
An IMU is an electronic device mounted on a platform. The IMU consists of individual sensors that report various information about the platform's motion. IMUs combine multiple sensors, which can include accelerometers, gyroscopes, and magnetometers.



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
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.



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: [1×1 accelparams]
        Gyroscope: [1×1 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:

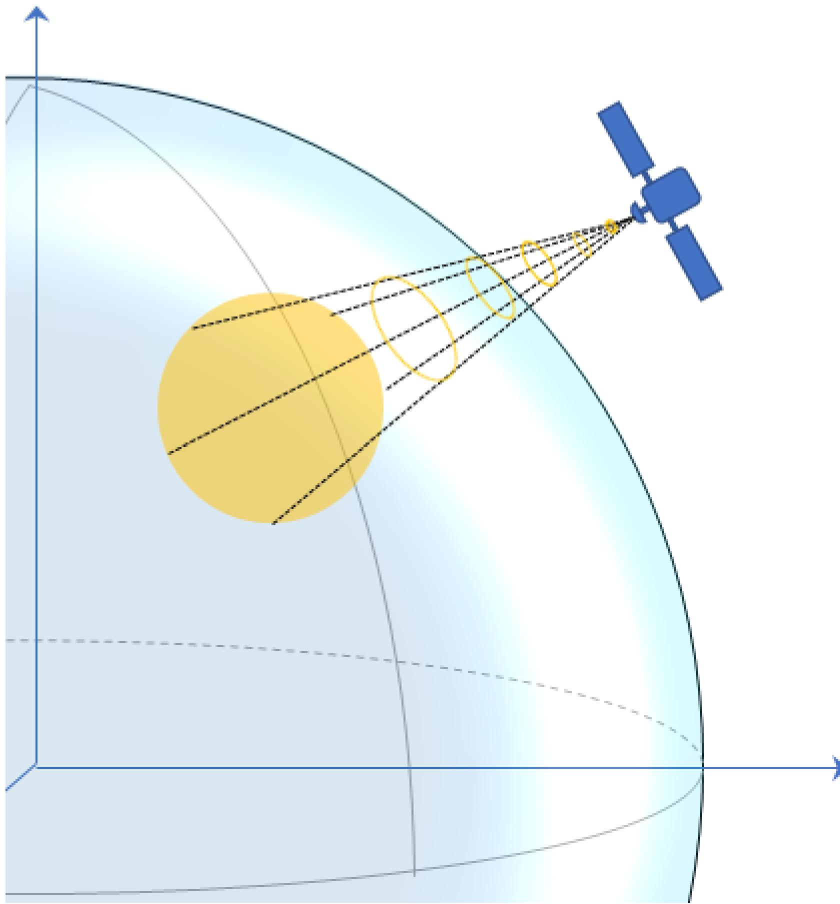
```
trueAcceleration = [1 0 0];
trueAngularVelocity = [1 0 0];
[accelerometerReadings,gyroscopeReadings] = IMU(trueAcceleration,trueAngularVelocity)
```

```
accelerometerReadings =

   -1.0000         0    9.8100


gyroscopeReadings =

     1     0     0
```

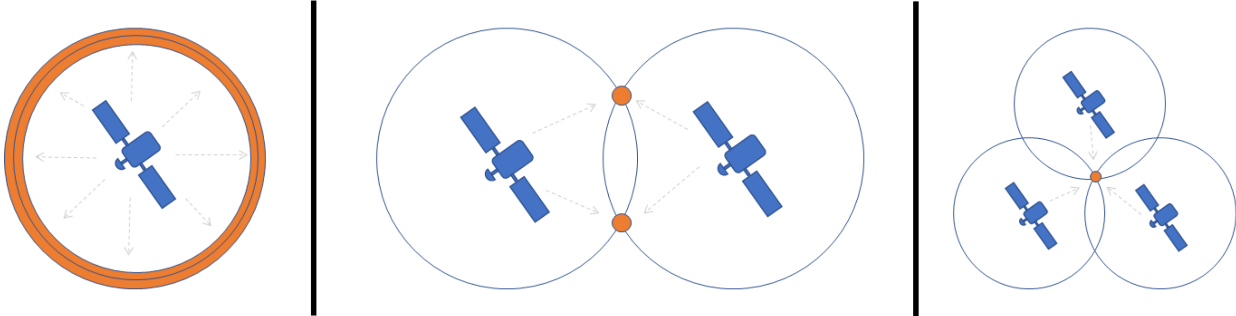
You can generate the ground-truth trajectories that you input to the IMU model using `kinematicTrajectory` and `waypointTrajectory`.

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

`gpsSensor` | `imuSensor`

External Websites

- https://www.gps.gov/systems/gps/

Occupancy Grids

In this section...

"Overview" on page 2-10

"World, Grid, and Local Coordinates" on page 2-11

"Inflation of Coordinates" on page 2-12

"Log-Odds Representation of Probability Values" on page 2-18

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 word 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.

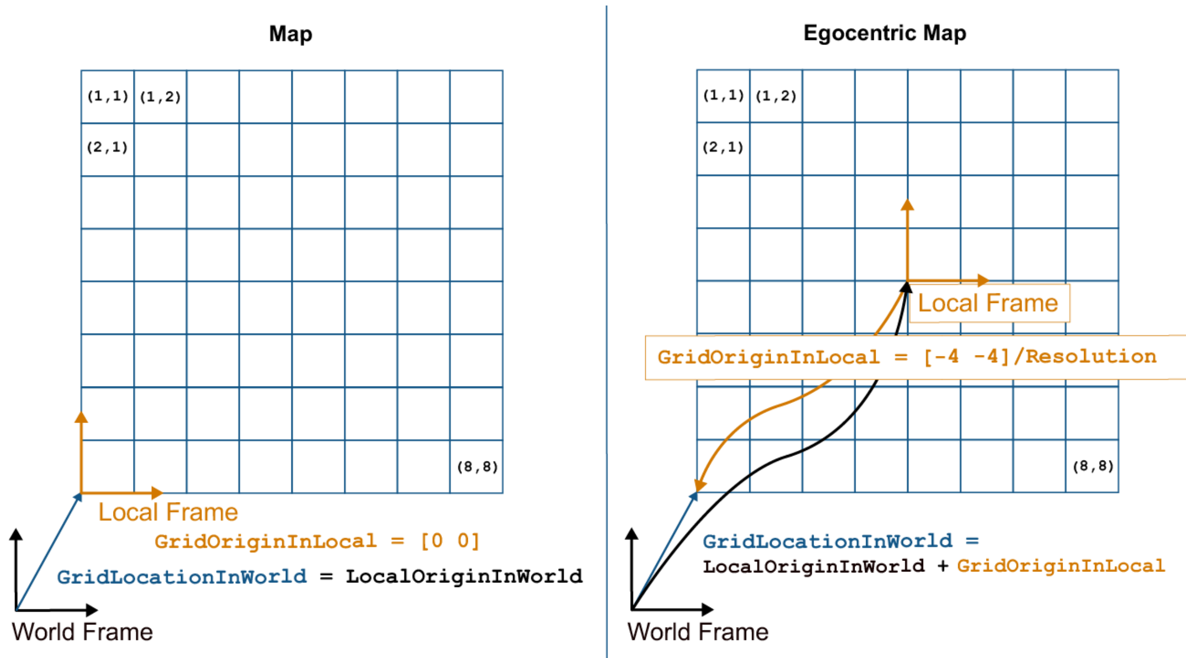
World, Grid, and Local Coordinates

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 1-161.

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

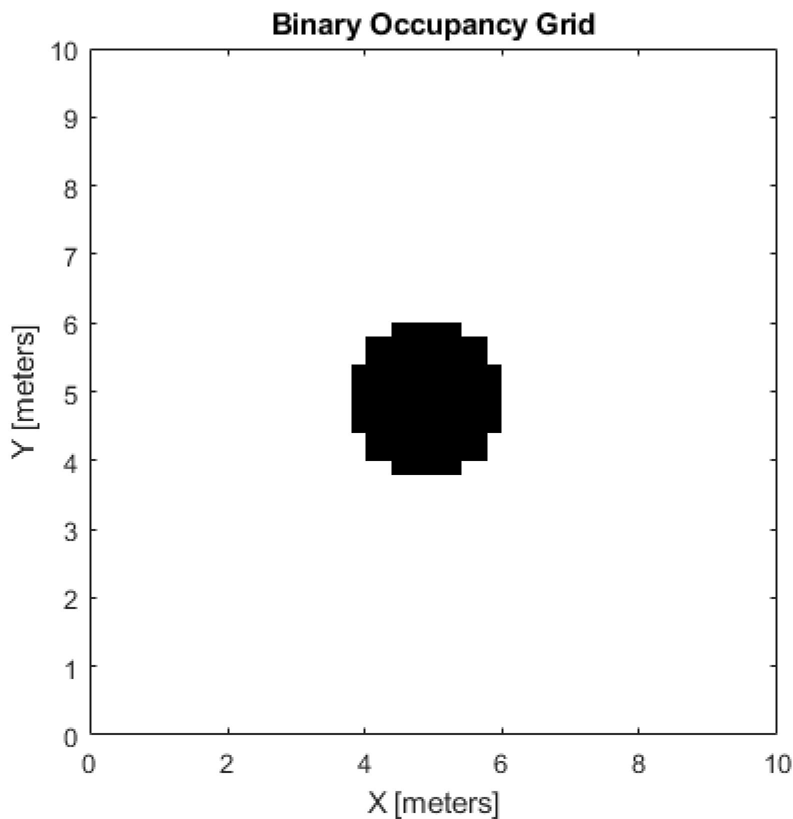
This example shows how to create the map, set the obstacle locations and inflate it by a radius of 1m. Extra plots on the figure help illustrate the inflation and shifting due to conversion to grid locations.

Create binary occupancy grid. Set occupancy of position [5,5].

```
map = binaryOccupancyMap(10,10,5);
setOccupancy(map,[5 5], 1);
```

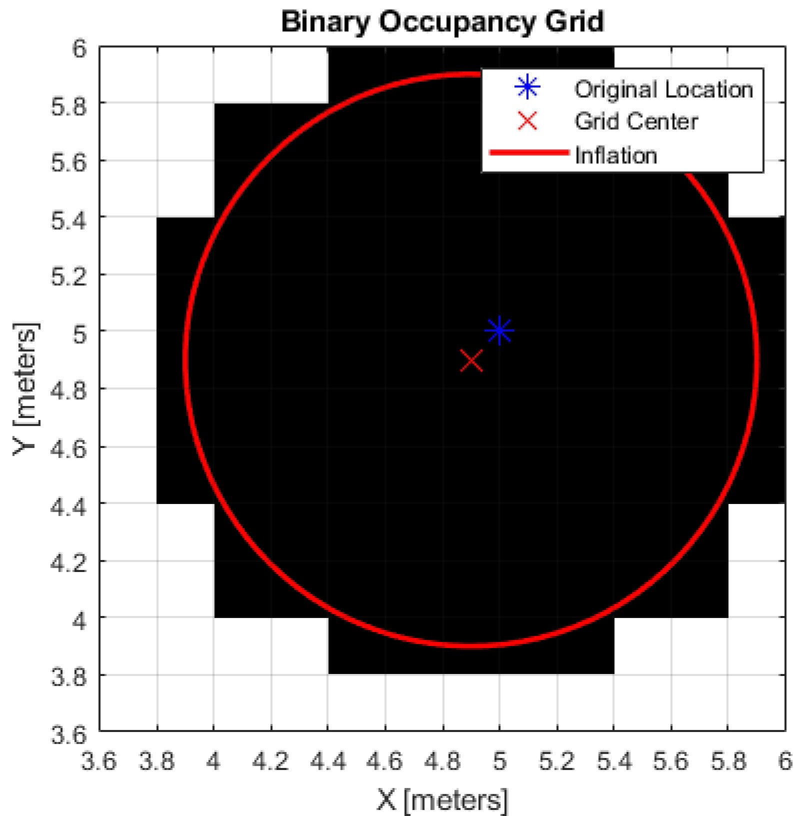
Inflate occupied spaces on map by 1m.

```
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

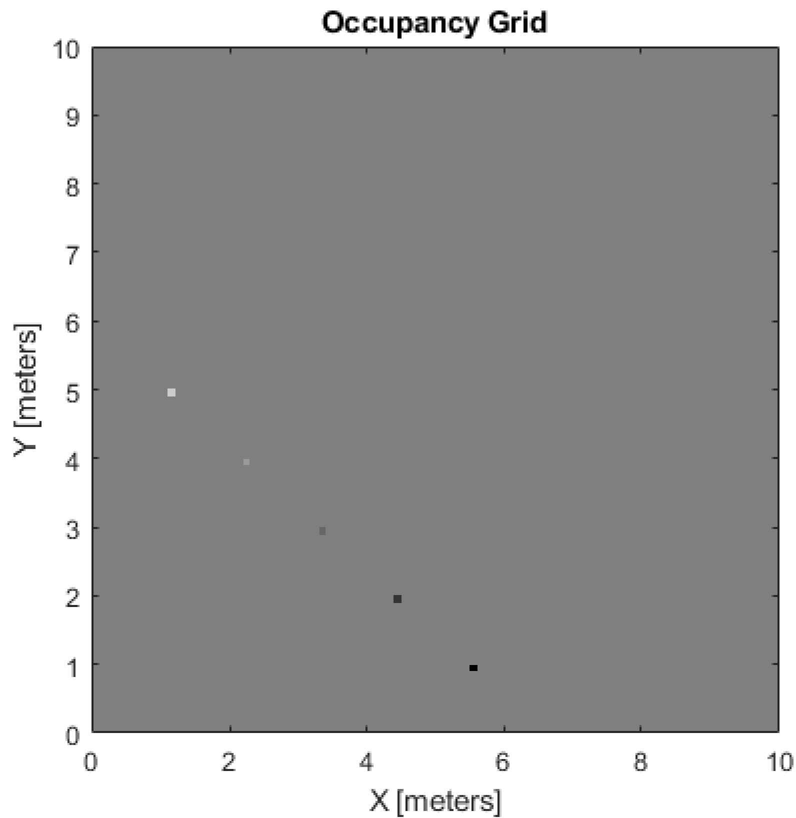
This example shows how the inflate method performs probabilistic inflation on obstacles to inflate their size and create a buffer zone for areas with a higher probability of obstacles.

Create a 10m x 10m empty map.

```
map = occupancyMap(10,10,10);
```

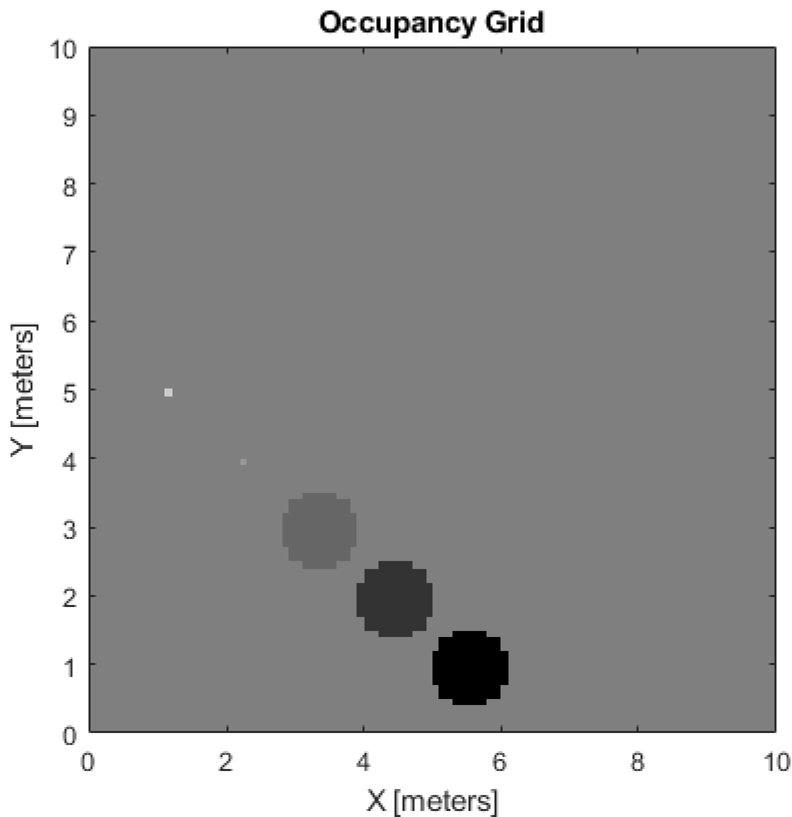
Update occupancy of world locations with specific values in `pvalues`.

```
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)
```

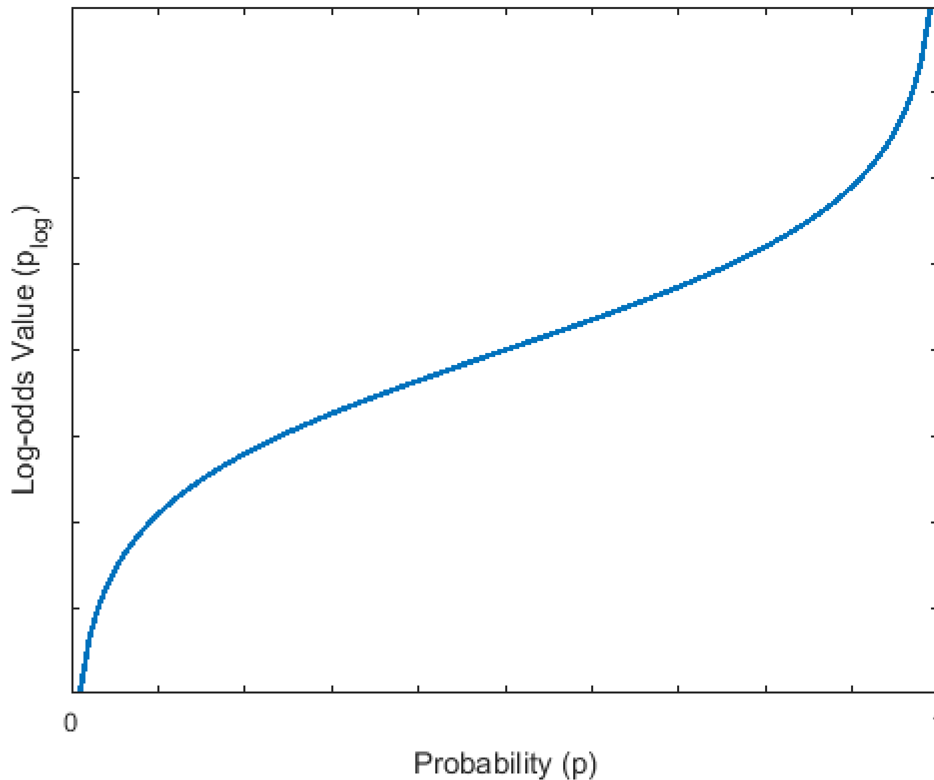


Log-Odds Representation of Probability Values

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:

$$p_{\log} = \log(\frac{p}{1-p})$$



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 1-161
- "Build Occupancy Map from Lidar Scans and Poses" on page 1-168

Execute Code at a Fixed-Rate

In this section...
“Introduction” on page 2-21
“Run Loop at Fixed Rate” on page 2-21
“Overrun Actions for Fixed Rate Execution” on page 2-22

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: 0.008895
Iteration: 2 - Time Elapsed: 1.005288
Iteration: 3 - Time Elapsed: 2.006203
Iteration: 4 - Time Elapsed: 3.010300
Iteration: 5 - Time Elapsed: 4.005049
Iteration: 6 - Time Elapsed: 5.004715
Iteration: 7 - Time Elapsed: 6.003483
Iteration: 8 - Time Elapsed: 7.003705
Iteration: 9 - Time Elapsed: 8.004860
Iteration: 10 - Time Elapsed: 9.003329
```

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: [1x20 double]
           NumPeriods: 20
```

```
        AveragePeriod: 1.0208
    StandardDeviation: 0.0431
          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: [1x16 double]
           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` | `rosrate` | `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.

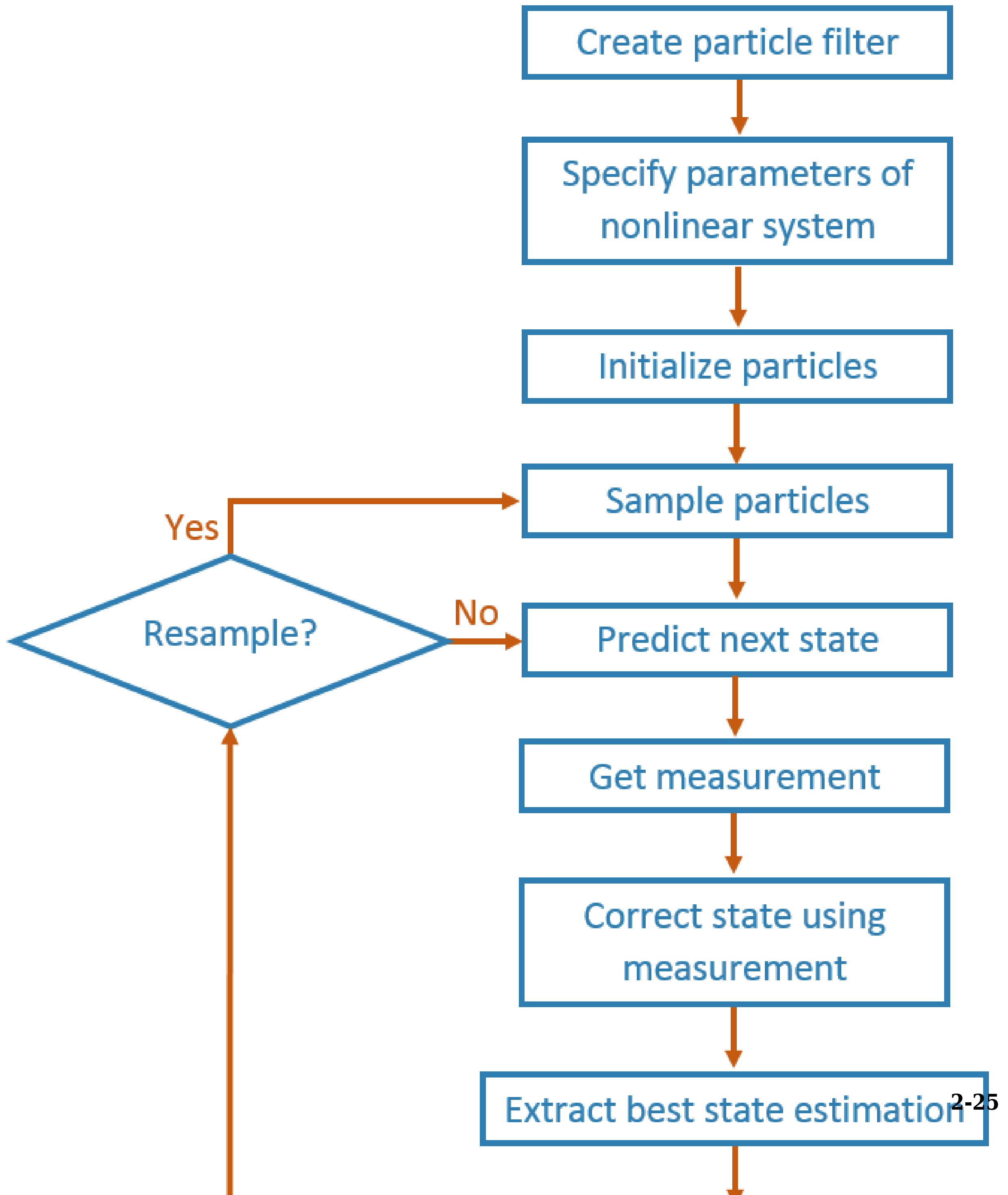
You can apply the particle filter to arbitrary nonlinear system models. Process and measurement noise can follow arbitrary non-Gaussian distributions.

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 2-29.

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
Specify parameters of nonlinear system
Initialize particles
Sample particles
Yes
Resample?
No
Predict next state
Get measurement
Correct state using measurement
Extract best state estimation

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 2-29.

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

`correct` | `getStateEstimate` | `initialize` | `predict` | `stateEstimatorPF`

Related Examples

- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Parameters” on page 2-29

Particle Filter Parameters

In this section...

"Number of Particles" on page 2-29
"Initial Particle Location" on page 2-30
"State Transition Function" on page 2-32
"Measurement Likelihood Function" on page 2-33
"Resampling Policy" on page 2-33
"State Estimation Method" on page 2-34

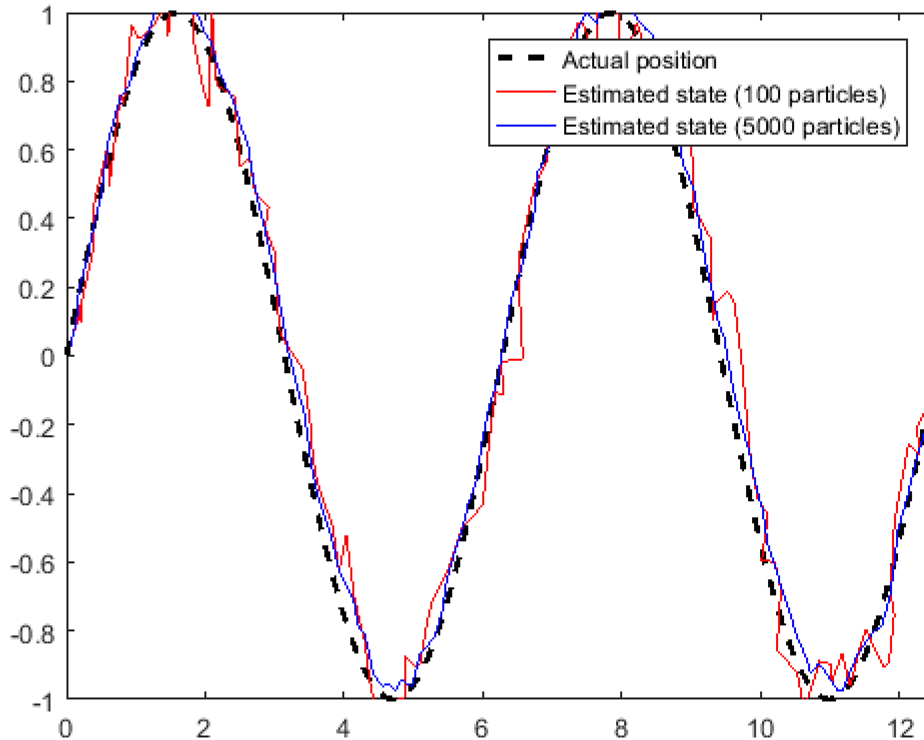
To use the `stateEstimatorPF` particle filter, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement 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 2-24.

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:

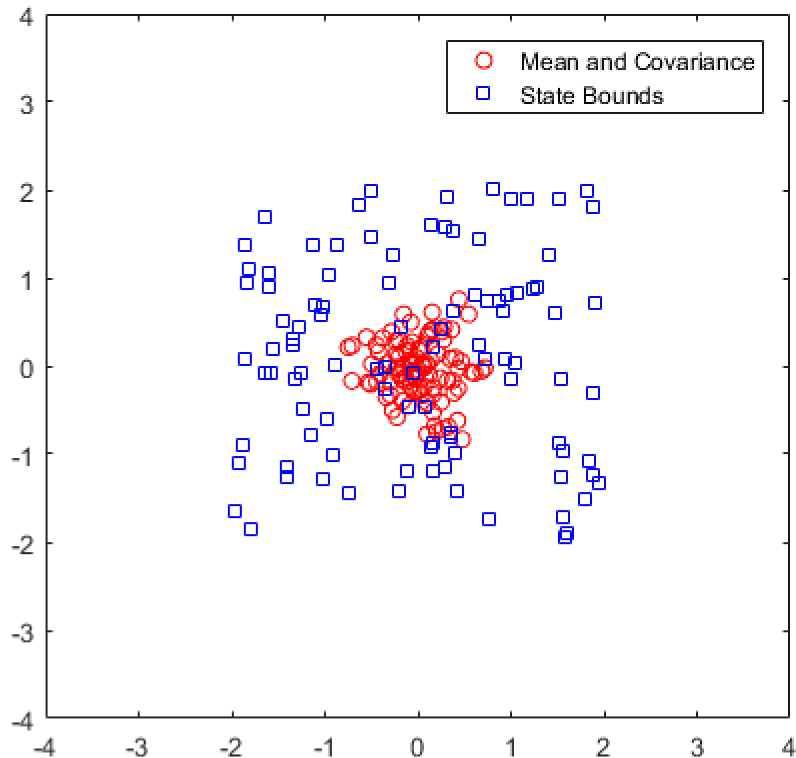
- Mean and covariance
- State bounds

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 2-24. 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 2-33 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 `resamplingPolicyPFresamplingPolicyPF` 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:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} \left(w^i\right)^2}$$

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} / `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

`resamplingPolicyPF` | `stateEstimatorPF`

Related Examples

- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Workflow” on page 2-24

Pure Pursuit Controller

In this section...

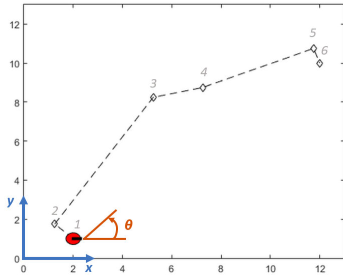
"Reference Coordinate System" on page 2-36
"Look Ahead Distance" on page 2-37
"Limitations" on page 2-38

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 `controllerPurePursuit` 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.

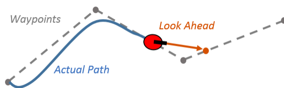
Reference Coordinate System

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 (theta) 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 *theta* value is the angular orientation of the robot measured counterclockwise in radians from the *x*-axis (robot currently at `0` radians).

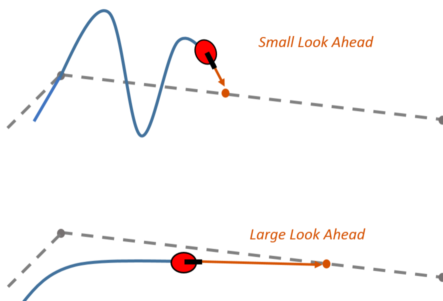


Look Ahead Distance

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

`controllerVFH` | `stateEstimatorPF`

Monte Carlo Localization Algorithm

In this section...

"Overview" on page 2-39
"State Representation" on page 2-40
"Initialization of Particles" on page 2-42
"Resampling Particles and Updating Pose" on page 2-44
"Motion and Sensor Model" on page 2-45

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:

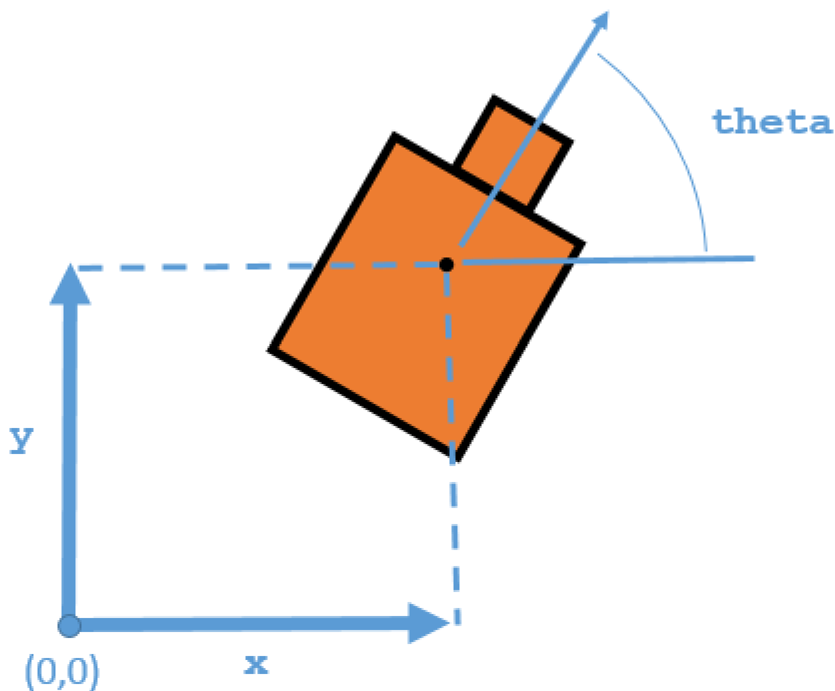
1. Particles are propagated based on the change in the pose and the specified motion model, `MotionModel`.
2. The particles are assigned weights based on the likelihood of receiving the range sensor reading for each particle. This reading is based on the sensor model you specify in `SensorModel`.
3. 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.

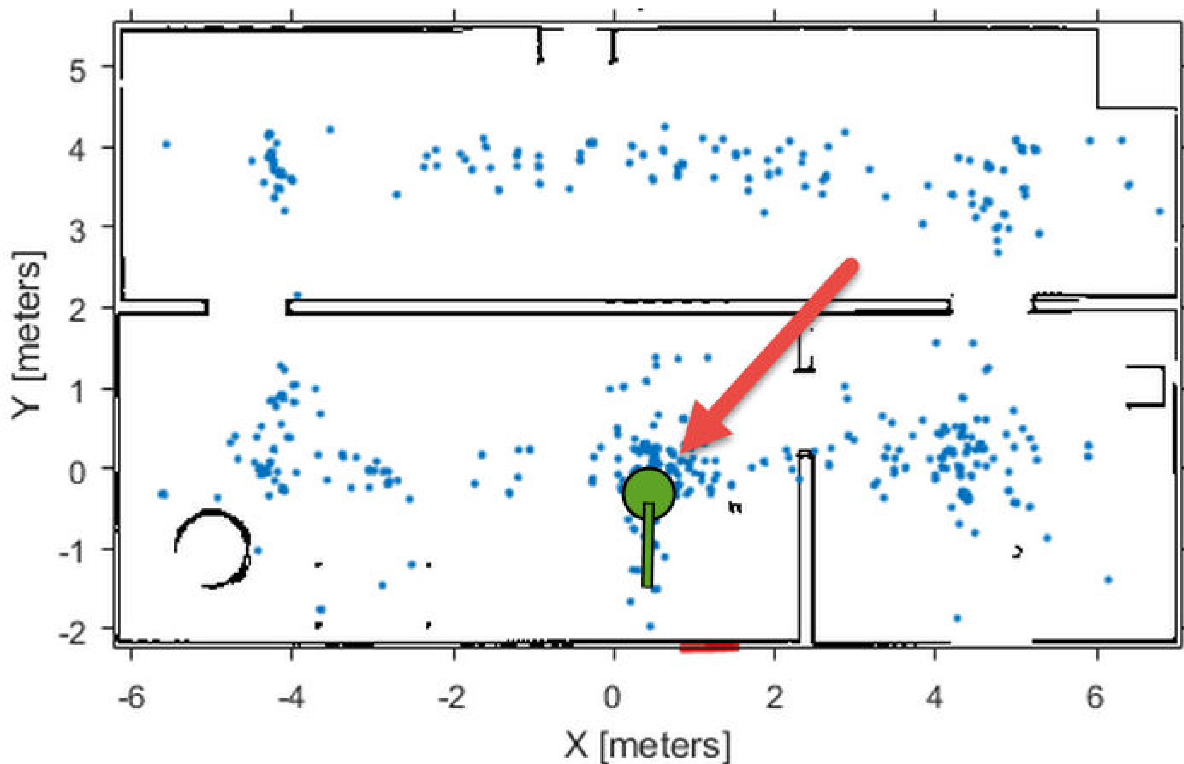
For more information on particle filters as a general application, see “Particle Filter Workflow” on page 2-24.

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 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`.

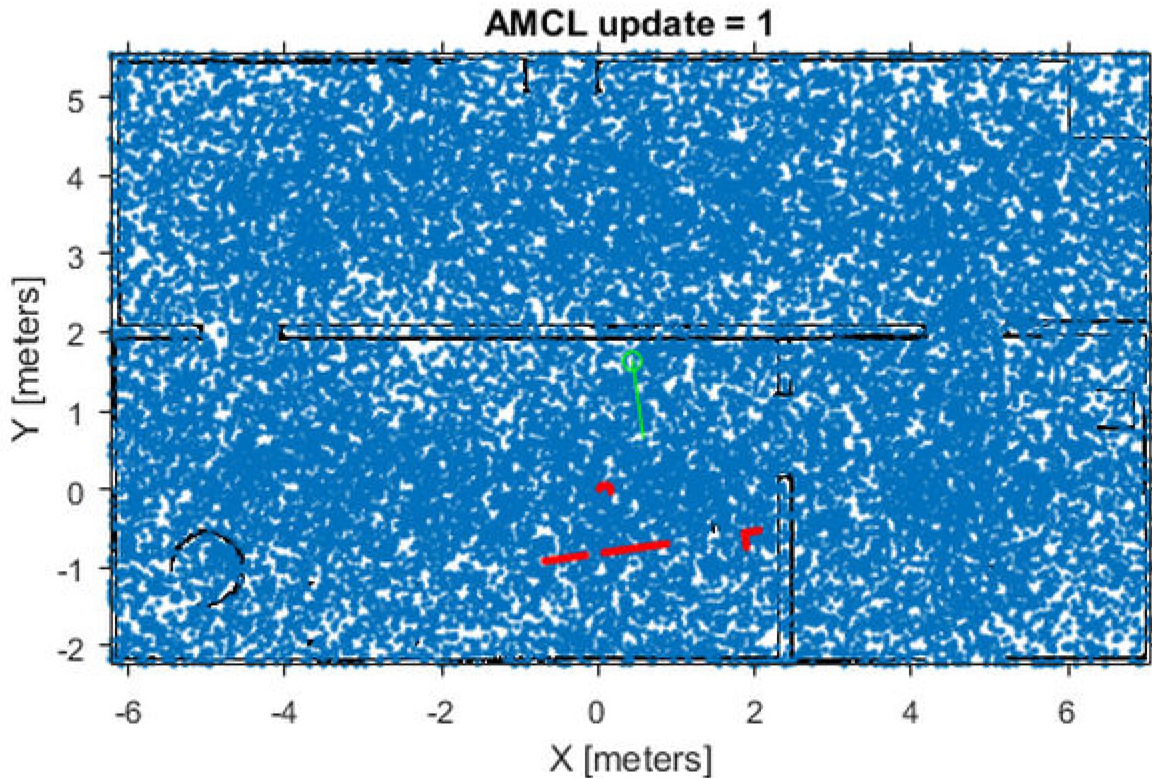
Initialization of Particles

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

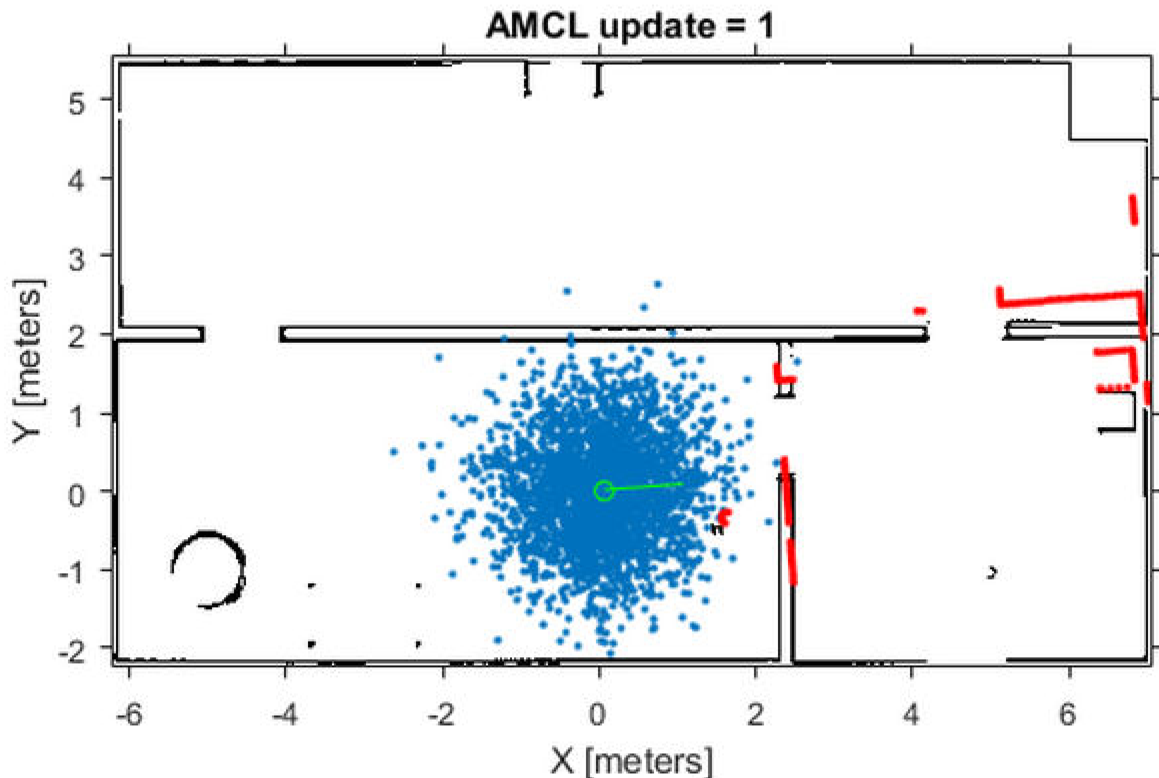
Particles must be sampled across a specified distribution. To initialize particles in the state space, you can use either an initial pose or global localization. With global localization, you can uniformly distribute particles across your expected state space (pulled from the `Map` property of your`SensorModel` 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 1-136 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 2-47. 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 2-29. 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 `likelihoodFieldSensor` 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.

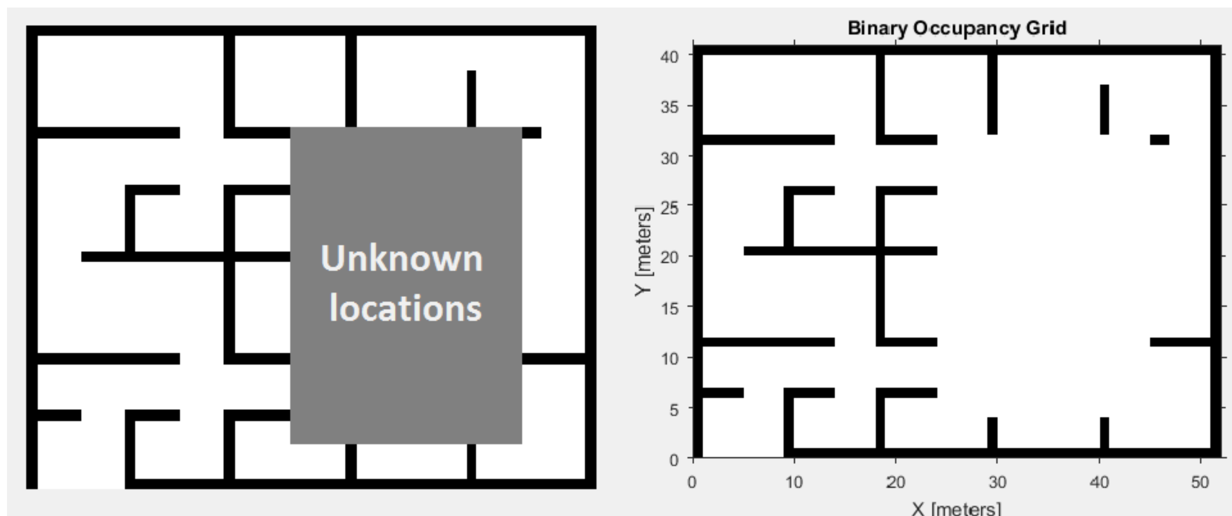
The range sensor properties are:

- `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.

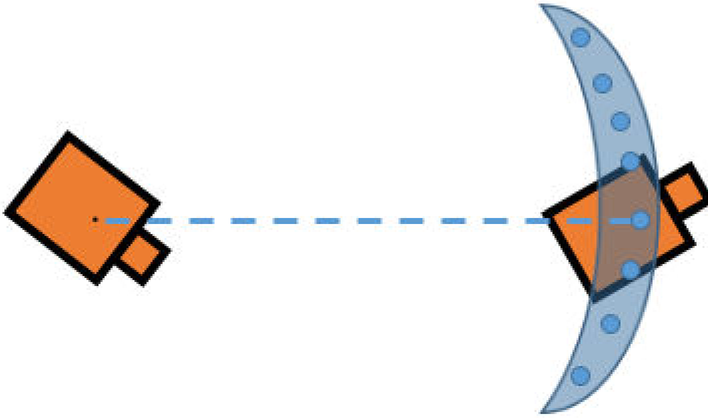
You can specify the error expected based on the motion of your robot as a four-element vector, `Noise`. These four elements are specified as weights on the standard deviations for [1]:

- Rotational error due to rotational motion
- Rotational error due to translational motion
- Translational error due to translational motion
- Translational error due to rotational motion

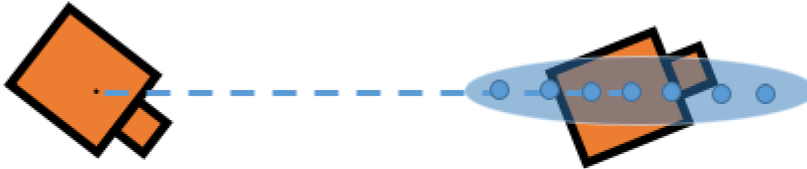
For differential drive robots, when a robot moves from a starting pose to a final pose, the change in pose can be treated as:

1. Rotation to the final position
2. Translation in a direct line to the final position
3. Rotation to the goal orientation

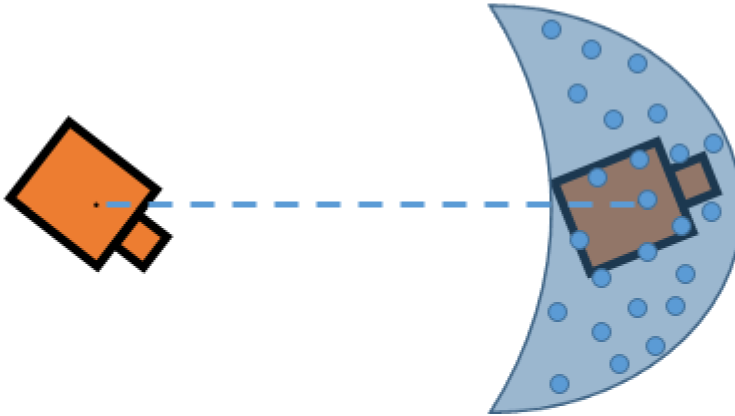
Assuming these steps, you can visualize the effect of errors in rotation and translation. Errors in the initial rotation result in your possible positions being spread out in a C-shape around the final position.



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

`likelihoodFieldSensor` | `monteCarloLocalization` | `odometryMotionModel`

Related Examples

- “Localize TurtleBot Using Monte Carlo Localization” on page 1-136

Vector Field Histogram

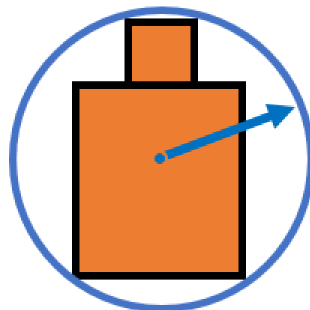
In this section...
“Robot Dimensions” on page 2-50
“Cost Function Weights” on page 2-52
“Histogram Properties” on page 2-53
“Tune Parameters Using show” on page 2-57

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.

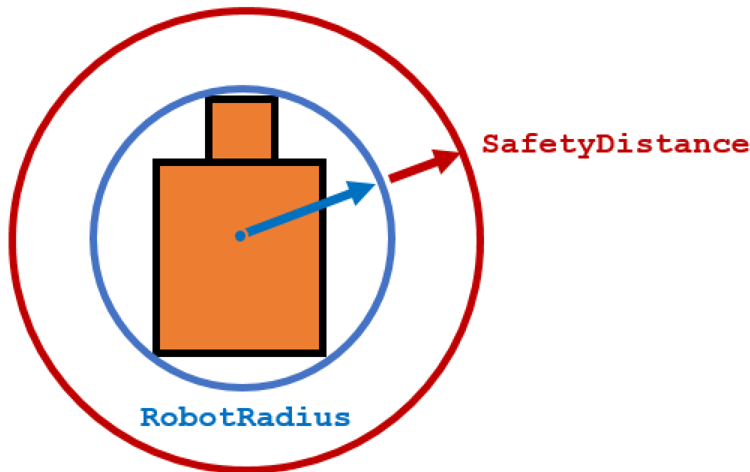
Robot Dimensions

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`.

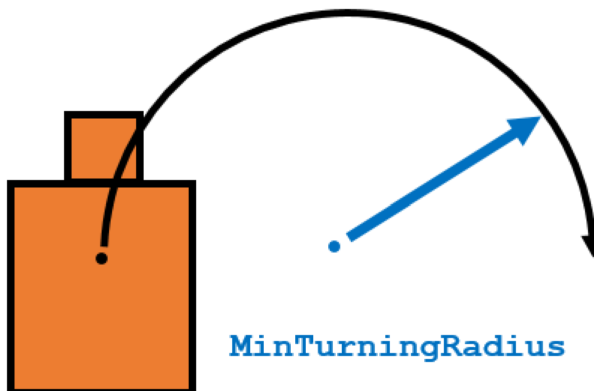
- `RobotRadius` specifies the radius of the smallest circle that can encircle all parts of the robot. This radius ensures that the robot avoids obstacles based on its size.



- `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.

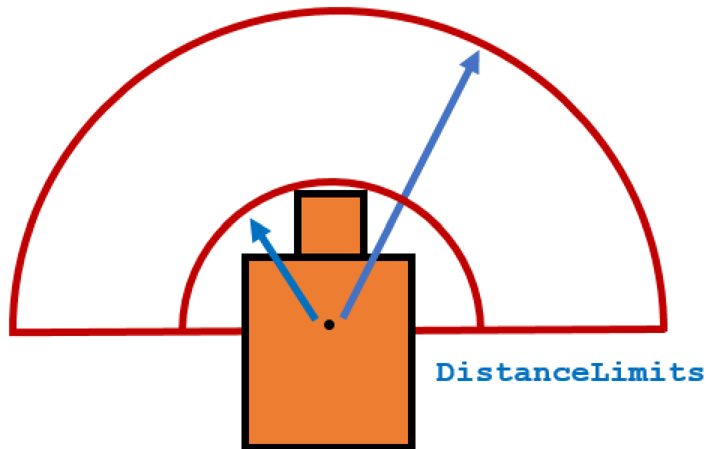


- `MinTurningRadius` specifies the minimum turning radius for the robot traveling at the desired velocity. The robot may not be able to make sharp turns at high velocities. This property factors in navigating around obstacles and gives it enough space to maneuver.



- `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.



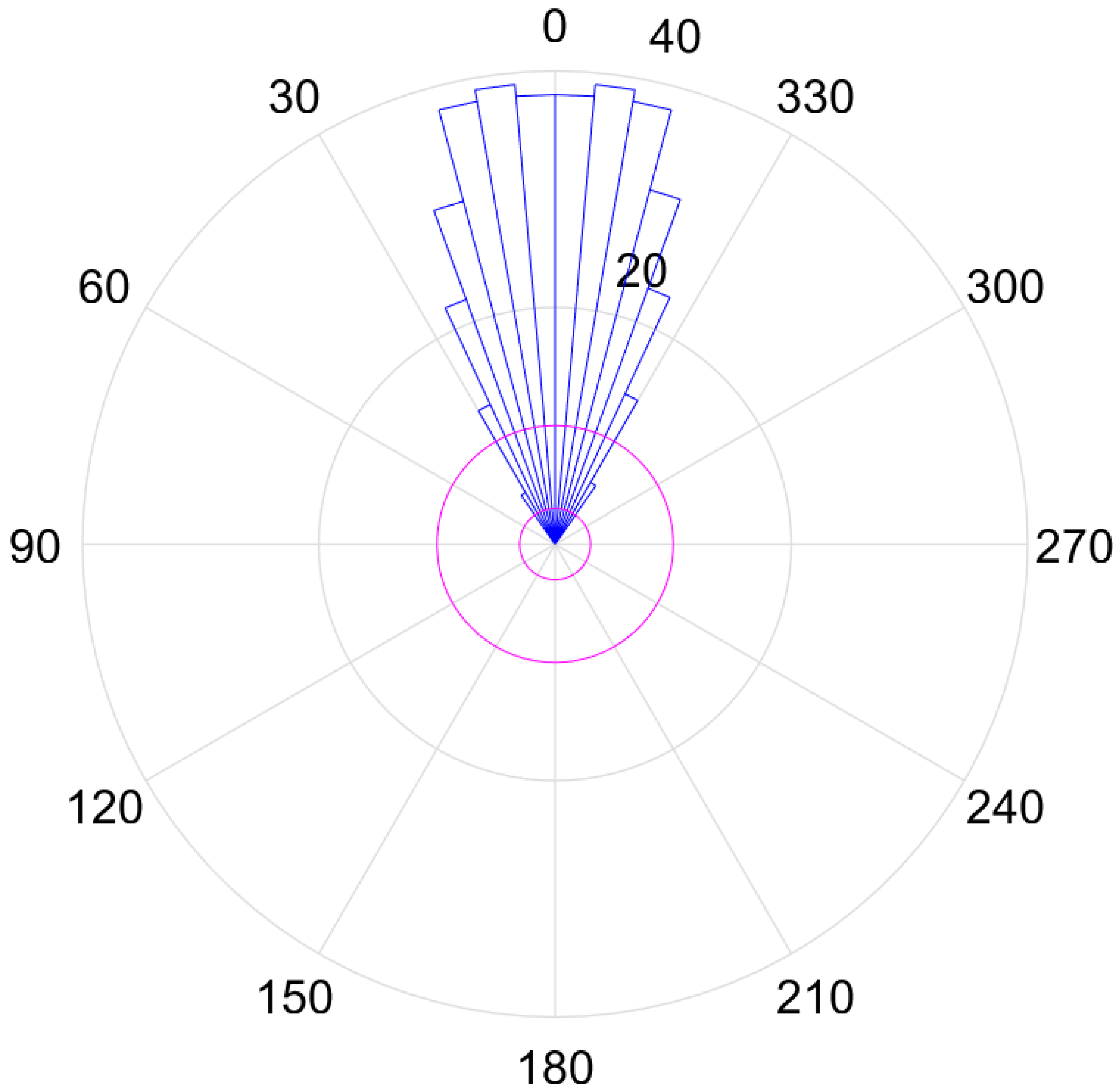
Note All information about the range sensor readings assumes that your range finder is mounted in the center of your robot. If the range sensor is mounted elsewhere, transform your range sensor readings from the laser coordinate frame to the robot base frame.

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

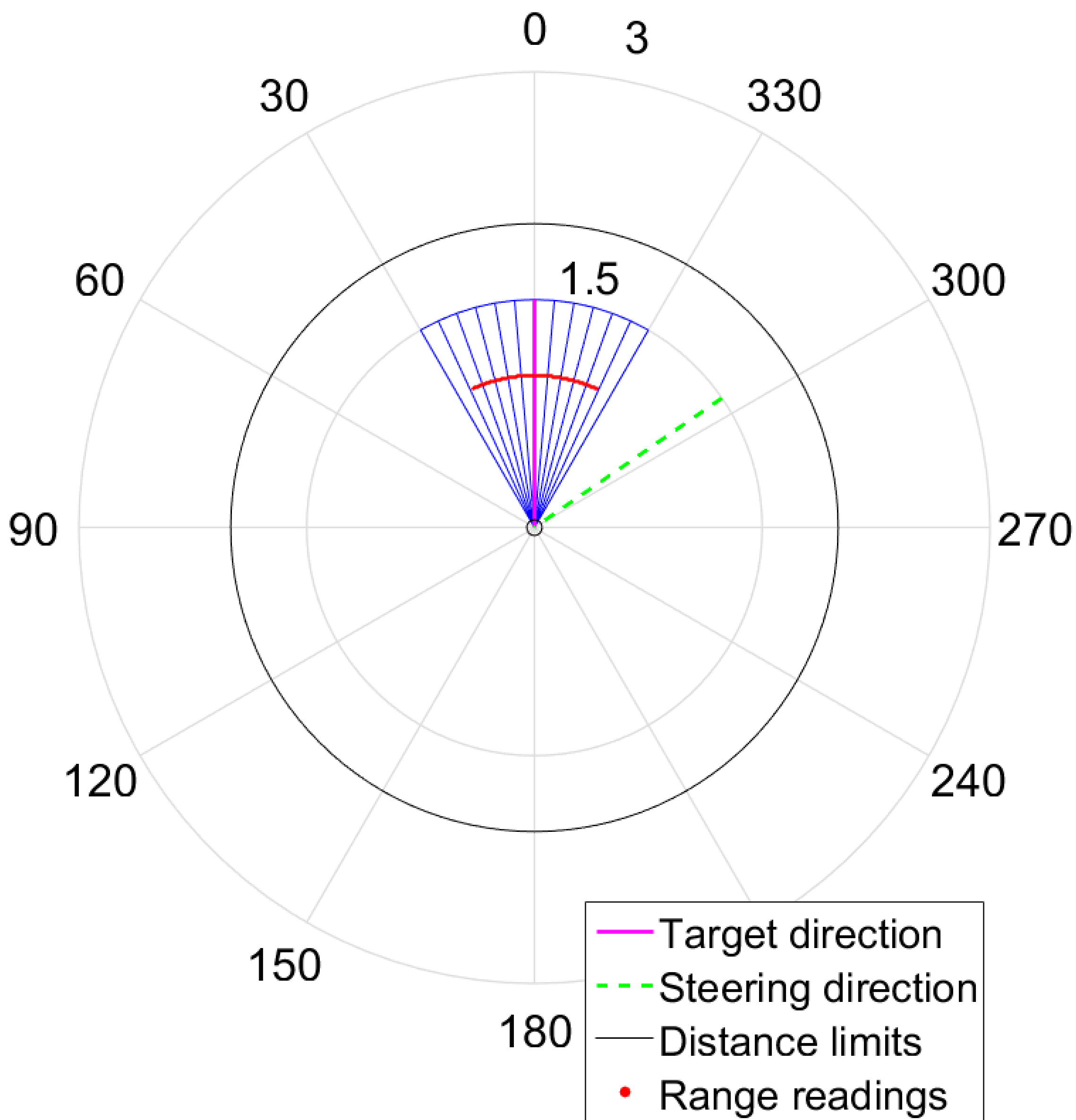
The VFH algorithm calculates a histogram based on the given range sensor data. It takes all directions around the robot and converts them to angular sectors that are specified by the `NumAngularSectors` property. This property is non-tunable and remains fixed once the `controllerVFH` object is called. The range sensor data is used to calculate a polar density histogram over these angular sectors.


Polar Obstacle Density
0
40
30
330
60
20
300
90
270
120
240
150
210
180

Note Using a small `NumAngularSectors` value can cause the VFH algorithm to miss smaller obstacles. Missed obstacles do not appear on the histogram.

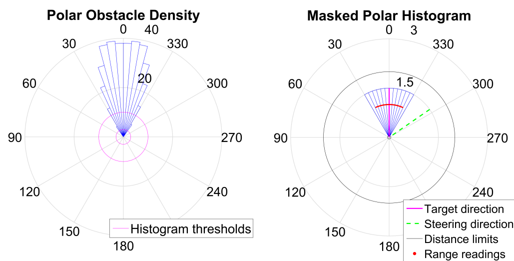
This histogram displays the angular sectors in blue and the histogram thresholds in pink. The `HistogramThresholds` property is a two-element vector that determines the values of the masked histogram, specified as `[lower upper]`. Polar obstacle density values higher than the upper threshold are represented 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.


Masked Polar Histogram
0
3
30
330
60
1.5
300
90
270
120
240
150
180
Target direction
Steering direction
Distance limits
Range readings

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 2-52 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')
```